```
"***********                                   ***********"
"***********        THE STEP PROCESSOR         ***********"
"***********                                   ***********"
"***********        BY JACK W. SIMPSON         ***********"
"***********    COMPUTATION RESEARCH GROUP     ***********"
"*********** STANFORD LINEAR ACCELERATOR CENTER ***********"
"***********      MENLO PARK, CALIFORNIA       ***********"
"***********                                   ***********"
"***********          PRINTED IN USA           ***********"
"***********                                   ***********"
```

ABSTRACT

A macro processor which implements both trigger and syntax macros is described. Trigger macros are implicitly called by the appearance of certain character sequences called triggers in the input text, while syntax macros are explicitly called from other macros. Local trigger macros may be defined in order to restrict the context in which they may be called. Macros of all types may be called recursively. The syntax of the macro patterns allows nesting of patterns, alternation, negation, optional matching, and iteration. Any of these constructs may be nested within any others to whatever degree desired.

The replacement text is generated by procedures which support integer, string, and symbol table data types; the appropriate operations for each of these; and conversion between data types. Replacement text can be directed to the input stream, output files, or other macros. The normal control statements such as IF-THEN-ELSE, WHILE, FOR, and GOTO are present. The language in which macro definitions are written is easily extensible and if desired can be made to conform to the syntax of whatever language is being processed. Except for a few limitations and a few extensions the capabilities of the processor are much the same as those specified in the IBM Language Point 2257 proposal.

The macro processor itself is distributed as an ANSI standard FORTRAN program and so can be implemented easily on any medium to large size computer.

Contents

1.0 Introduction

The STEP processor belongs to the class of text processing programs known as macro processors. It is designed primarily to allow user defined languages or language extensions to be easily and quickly implemented. Text processing by STEP can be thought of as being table driven, with the data elements contained in the table being called macros. Each of these macros can be thought of as a formula or rule for transformations to be performed on the input data being read by the processor. This input data could, for example, be the source of a computer program which is written in a specialized language and must be transformed into text suitable for input to a specific compiler or assembler.

The macros, which are user defined, can be thought of as the software which directs the hardware (the STEP processor) in the performance of its task. A simple example of the definition of a macro and the actions that it would cause the processor to take will now be described.

When no macros are defined, the processor simply reads text from its input file and writes that text unchanged to its output file. A simple text replacement macro could be defined by the appearance in the input of the string

```
MACRO TRIGGER: 'INTEGER'; RESCAN 'REAL'; END MACRO;    (1.1)
```

The string `MACRO TRIGGER:` can for the present be thought of simply as an indicator to the processor that a macro definition follows. When this or any other macro definition is encountered in the input it is recognized by another macro, which removes it from the input stream, possibly translates it into an intermediate form, and passes it to internal compilers. The internal compilers translate the macro definition into object code which is stored in the processor's tables. The macro definition in this case consists of three statements each of which is terminated by a semicolon. The significant portion of the macro is its first and second statements, while the third serves to indicate the end of the definition. The first statement of a macro definition is always a pattern matching statement and indicates the type of text in the input that is to be transformed. In this case it is the string `INTEGER`. This string is delimited by apostrophes in order to separate it from the various control symbols and other entities that might appear in the first statement of a more complex macro definition. The keyword `RESCAN` in the second statement indicates that the string which follows it (`REAL`) is to replace any occurrences of the pattern in the input. Thus the input

```
 INTEGER X,Y,Z;
 MACRO TRIGGER: 'INTEGER'; RESCAN 'REAL'; END MACRO;   (1.2)
 INTEGER A(1),B,C(INTEGER);
```

will appear in the output as

```
 INTEGER X,Y,Z;                                        (1.3)
 REAL A(1),B,C(REAL);
```

The first occurrence of the string `INTEGER` is not transformed since it appears before the macro is defined. Actually the output of the processor may have slight differences in formatting from (1.3), but for the present discussion these are unimportant.

It should be emphasized that the macro defined in (1.2) is an extreme simple case of the types of macros that can be defined using STEP.

1.1 Summary of Contents

The primary purpose of this manual is to instruct the reader in how to define and use macros with the STEP processor. Examples will be used whenever possible to help clarify in the reader's mind the various rules and concepts as they are introduced. While at first the examples given may seem trivial, as more of the features of the processor are defined and made available for their use, the examples will take on a more practical flavor. Many of these examples will demonstrate ways to extend an existing language. Unless otherwise stated, the base language that is extended is similar to FORTRAN, except that it is free field, with statements being separated by semicolons.

2.0, “Macro Processing” on page 4 begins with a discussion and examples of simpler types of macros and then attempts to motivate the need for some of the more complex ways in which STEP macros can be used. Explanations given will aid the reader's intuition in understanding the examples in this section. This understanding can then be completed by the more detailed discussions of macro definition and usage given in later sections. The two types of STEP macros, syntax macros and trigger macros, are introduced and a short description of their behavior and some possible uses complete this section.

3.0, “Overview of the STEP Processor” on page 14 presents an overview of the STEP processor. First its structure is briefly outlined and then the method by which it reads, transforms, and writes text is explained in a little more, although not yet complete, detail. The various I/O files used by the processor are also briefly described.

4.0, “Input Reader” on page 16 discusses the input readers and several miscellaneous details of the STEP processor which could not conveniently be explained elsewhere.

The next two sections constitute a reference manual for STEP macro pattern definitions. The first of these deals with the elements from which all macro patterns are constructed: the pattern string and the syntax macro call. 6.0, “Macro Pattern Definition: Pattern Construction” on page 29 defines the various control symbols used in constructing complex patterns from these elements. A detailed account of an example matching process is also given.

The next three sections are a reference manual for the replacement procedure compiler. 7.0, “Macro Replacement Procedure Definition: Introduction, Data Types, and Expressions” on page 37 discusses data types and the expressions which can be formed from them. 8.0, “Macro Replacement Procedure Definition: Statements” on page 46 discusses the statements in the replacement procedure language. The next section completes the discussion of the compiler with intrinsic functions, listing format, and a few suggestions on how the compiler should best be used.

10.0, “Macro Definition” on page 63 completes the formal discussion of macro definition with rules for defining and using local trigger macros. Also included is a discussion and example of using STEP macros for extending the macro definition language itself. The section ends with a discussion of the protect option, which allows the invocation of certain classes of STEP macros to be temporarily inhibited.

The text that has been transformed by the matcher is formatted for output by the output processor, which is described in 11.0, “Output Processor” on page 67. The STEP output processor is somewhat unusual in that it effectively performs some limited lexical scanning operations on the text that is input to the macro processor.

12.0, “Techniques for Macro Definition and Use” on page 69 is a relatively unstructured collection of hints and techniques for writing and using macros. Finally, the conclusion in 13.0, “Conclusion” on page 74 describes a number of changes that the processor has yet to undergo.

This manual is directed toward the macro writer who will be using the STEP processor to implement or extend computer programming languages and not the end user of a language which is defined by a particular set of STEP macros. The macro writer is expected to be an experienced programmer with knowledge of more than one high level language. Depending upon the complexity of his task, a knowledge of some of the techniques of compiler construction may also be helpful. This is not to say that users of languages implemented by the macro writer cannot write macros, because even though the full capabilities of the STEP processor may be more than they wish to learn, the macro writer, using the ability of STEP macros to process the definitions of other STEP macros, can easily implement a simpler macro definition language to accompany the language being defined or extended. Such a macro language would perhaps use only a small subset of the full capabilities of the macro definition language and could be made to have a syntax similar or identical to that of the language being defined or extended.

Many existing general purpose macro processors are somewhat elegant in the simplicity of their macro definitions and as a result of this require complex combinations of interacting macros in order to perform some tasks. STEP makes no claims to elegance or simplicity; it does, however, claim practicality. There are many different statements, control symbols, and matching rules that must be learned in order to use the processor effectively, and the functioning of the recursive matching process is at times difficult, although not impossible, to understand. Once the user has learned to use the processor, however, he will find that many of the complex things that he wishes to do can be done in a straightforward manner without the necessity of resorting to various tricks or complex constructions of macros.

1.1.1 Notation

When portions of a macro definition, keywords, or characters that are used in the coding of macro definitions or input text for the processor are included directly in the text of this manual they will normally be distinguished easily enough and not need to be delimited by quotation marks or any other special symbols. In those few instances where confusion could arise, however, quotation marks will be used as delimiters and should not be considered as part of the actual text that would be passed to the processor.

2.0 Macro Processing

2.1 Trigger Macros

In its simplest form, a macro processor is a program which, upon recognition of certain strings in an input text stream, substitutes for these strings corresponding strings called replacements. The recognition, or triggering, process is accomplished by matching a prefix string of the current input stream against patterns, which are text strings stored in the processor. To increase the flexibility of this process the patterns may be allowed to contain formal arguments, so that portions of the input text might be required to match identically those parts of a pattern before and after the formal argument, while another portion of the input, called the actual argument, has only a few restrictions on its content.

For example, a macro for the STEP processor could have pattern

```
ADD' A1:BAL 'TO' A2:BAL 'AND STORE INTO' A3:BAL ';'     (2.1)
```

where the text delimited by apostrophes must be matched literally and the BAL terms denote formal arguments which are distinguished from each other by the labels A1, A2, and A3. The occurrence in the input text of

```
            ADD 32 TO VAR AND STORE INTO INT;           (2.2)
```

would match the above pattern and cause its corresponding replacement to be produced. The production of the replacement can be governed by a template of the form

```
                  A3 '=' A1 '+' A2 ';'                  (2.3)
```

where items delimited by apostrophes can be thought of as character string constants and items not so delimited as character string variables representing the actual arguments. The replacement is then formed by substitution of the actual arguments 32, VAR, and INT, for the occurrences in the template of A1, A2, and A3, which correspond to the first, second, and third formal arguments occurring in the pattern. The replacement for the above example would then be

```
                     INT = 32 + VAR ;                   (2.4)
```

In order to define this macro one would then write

```
MACRO TRIGGER:
  'ADD' A1:BAL 'TO' A2:BAL 'AND STORE INTO' A3:BAL ';';
  RESCAN A3 '=' A1 '+' A2 ';' ;                     (2.5)
  END MACRO;
```

The pattern appears as the first statement of the definition while the replacement template appears in the second. The necessity of delimiting the pattern and parts of the replacement template by apostrophes is in part demonstrated by the fact that the semicolons which appear in both the pattern and in the replacement template must not become confused with the semicolons which terminate statements in the macro definition. The macros (2.5) and (1.1) are called trigger macros because their activity is triggered by the appearance of strings in the input stream which match their patterns.

2.1.1 Elementary Macro Writing - Examples

A few examples of the practical use of some of the macro concepts thus far introduced are now given for the benefit of the reader who is new to macro processing. The more experienced reader will find these examples similar to those arising from common usage of other macro processors and may wish to skip this section.

The first example illustrates how common declarations in a large FORTRAN program can be centralized by the use of macros of the form

```
MACRO TRIGGER:
    'COMMON/PLOT1D/;';
    RESCAN 'COMMON/PLOT1D/VAR(3,6,32),IDENT(8,8),'
           'FCENTR(2,20,36),QLOG(100);'              (2.6)
           'LOGICAL QLOG; INTEGER VAR,IDENT;' ;
    END MACRO;
```

The FORTRAN programmer may define this macro, and others like it, at the beginning of a large group of FORTRAN routines in order to save writing the same common declarations again and again in each routine, and indeed, to insure that the common declarations in each routine are the same. If a program modification requires that a common declaration be changed, then the change need only be applied to one place in a macro and it will be propagated throughout the rest of the program. For really large programs, macros like the following could then be defined

```
MACRO TRIGGER:
    'INCLUDE GRAPHICS COMMONS;';
    RESCAN 'COMMON/PLOT1D/; COMMON/PLOT2D/;'       (2.7)
           'COMMON/CHARS/;  COMMON/SCOPEC/;';
    END MACRO;
```

Note that the text produced by this macro is placed back into the input stream where it is further converted by macros like (2.6). When writing a graphics subroutine the appropriate common blocks can now be declared by one statement.

The second example illustrates how a macro which makes use of a single formal argument can be used to provide a shorthand notation for a common type of program statement. In some programs it is often necessary to increment or decrement a variable by one. This is normally done by coding a statement like

```
ISCOPE(IARRAY,JARRAY*64+38,KARRAY*32-16) =              (2.8)
       ISCOPE(IARRAY,JARRAY*64+38,KARRAY*32-16) + 1;
```

This extreme example suggests that writing the ISCOPE array with all of its subscript expressions only once would lessen the chance of a coding error, not to mention saving wear and tear on the programmer. The new syntax might be

```
+ ISCOPE(IARRAY,JARRAY*64+38,KARRAY*32-16);      (2.9)
```

So a statement beginning with a '+' (or '-') means that the variable in the statement must be incremented (or decremented) by one. The macro implementing the increment statement could be written

```
MACRO TRIGGER:
    ';+' BAL ';';
    RESCAN ';' BAL '=' BAL '+1;';          (2.10)
    END MACRO;
```

Note that the terminating semicolon of the previous statement is used to ensure that the '+' operator is the first character of the statement that it appears in. Also, since only one formal argument appears in the pattern of (2.10), it does not need to be labelled.

2.1.2 Extensions Needed for Macro Language Processing

Suppose now that the macro writer has produced a special purpose language having statements similar to (2.2) for a certain user community. The statements are translated by a macro processor into a FORTRAN like base language having statements of the form (2.4). The members of the user community have decided not to learn FORTRAN, but instead prefer to program entirely in the macro defined special purpose language. If one of these programmers were to code the incorrect statement

```
ADD A TO B AND STORE INTO 38;            (2.11)
```

it would be translated without complaint by (2.5) into the incorrect base language statement

```
38 = A + B ;                             (2.12)
```

Later, upon reaching this statement, the base language compiler will return to the user the following diagnostic:

```
STATEMENT SYNTAX ERROR:     38=A+B;          (2.13)
```

Since the object (FORTRAN) code, with which the user is unfamiliar, appears in the diagnostic, it may be difficult to trace the error back to the faulty source statement. Indeed, the translation process for a single statement in some macro defined languages may involve a number of macros, some of which may be considerably more complex than (2.5), so that tracing an error in the macro produced object code back to the source may be impossible for one who is familiar with neither the base language nor the translating macros.

While the macro processor can simply insert the actual arguments into various parts of the replacement as in the example above, it may also have the ability to make decisions as to the nature of the arguments and formulate its replacement accordingly. For example, if the actual argument represented by A3 is inspected and found to be a number rather than a variable, instead of the normal replacement action an error message could be sent to a diagnostic file. In order to direct the operation of this and various other replacement functions available in such an extended processor, the replacement template must instead become a replacement procedure with perhaps arithmetic and logical as well as text handling facilities.

If the type of text acceptable as an actual argument could be appropriately restricted the above problem could also have been avoided. For example, a more specific pattern could be written:

```
'ADD' A:BAL 'TO' B:BAL 'AND STORE INTO' ID ';'      (2.14)
```

where ID is a new kind of formal argument whose actual argument must be an identifier. Nearly anything was allowed to be the actual argument corresponding to BAL. So while (2.2) would still match the above pattern, (2.11) would not, and so would be passed as written by the user to the base language compiler which would now produce the diagnostic

```
STATEMENT SYNTAX ERROR:   ADD A TO B AND STORE INTO 38;  (2.15)
```

which the user will more readily understand.

2.2 Syntax Macros

Because the ID formal argument is so specific as to what its actual argument must be, it can be thought of as a pattern in its own right, which in this case generates replacement text identical to that which it matched. So the pattern in (2.14) can be said to have an explicit call to the ID macro from which it receives the replacement text that it will use as its third actual argument. In addition to macros which match identifiers, it would also be useful to have macros with patterns capable of matching other syntactic elements such as integers or even more complex constructs such as arithmetic expressions. These syntax (See 13.2, “References” on page 74, item 1) macros would be activated by explicit call from other macros, as in (2.14), and would return their replacement text to the calling macro.

The two types of macros basic to the STEP processor have now been introduced. The first of these is the trigger macro, examples of which are (1.1) and (2.5). A trigger macro is implicitly called by the appearance in the input of a string matching the first quoted string in its pattern. After its pattern matches, the macro's remaining statements generate the replacement text, which is normally returned directly to the input stream where it replaces the string which was matched. The processor's scan of the input stream then resumes at the beginning of this replacement. The second type is the syntax macro. Each syntax macro is given a name (for example, ID) and must be explicitly called by the appearance of that name in the pattern of another syntax or trigger macro. After the pattern of the syntax macro matches the input the macro's remaining statements generate the replacement text which is usually returned as an argument to the calling pattern. The statements of a trigger macro definition must be preceded by the text ″MACRO TRIGGER:″ while the syntax macro definition statements are preceded by ″MACRO SYNTAX: name″, where name is an identifier which will be used to explicitly invoke the new macro. To better understand the distinction between these two types of macros the following two examples are given.

The base language is to be extended by a statement which will allow any scalar variable to be negated. The NEGATE macro is defined as

```
MACRO TRIGGER:  'NEGATE' ID ';';
      RESCAN ID '=-' ID ';';              (2.16)
      END MACRO;
```

The identifier MACRO indicates to the processor that a macro definition follows. The next identifier tells what type of macro is being defined, and, as mentioned earlier, the final statement signals the end of the definition. For the moment the ID syntax macro will be assumed to exist, to have a pattern which matches any identifier, and to produce that same identifier as its replacement text. Note that the appearance of ID in the pattern of (2.16) is a call to the ID syntax macro, while its appearance in the replacement template is as a string variable which has been initialized to the text matched by the ID macro. The input text to be scanned consists of the statements

```
CALL FCN(NEGATE,VAR);               (2.17)
NEGATE VAR;
```

The scan of the input text begins under the control of the processor, which causes the scan to proceed on an atom by atom basis. For the present an atom is defined to be any identifier, integer (sequence of digits), or single non-alphanumeric character. The atoms in the first line of (2.17) are then

```
CALL    FCN    (    NEGATE    ,    VAR    )    ;
```

The atom CALL is first scanned, and when no matching macros are found is passed to the output file. In the same way the atoms FCN and "(" are scanned and passed to the output file. When the processor begins to scan the next atom it passes control of the scanning process to the NEGATE macro which matches character by character the string NEGATE in its pattern with the same sequence of characters in the input. The NEGATE macro then passes control of the scanning process to the ID syntax macro which finds a comma instead of the beginning of an identifier and so fails to match the input. The ID macro then returns control and word of its failure to the pattern of the NEGATE macro, causing it to fail also. The NEGATE macro then returns control of the scan to the processor which restores the environment to what it was before the NEGATE macro was called. The processor then passes scan control to other trigger macros which have not yet been tried and when these fail passes the atom NEGATE to the output file. After the next four atoms are scanned and passed to the output file the processor again encounters the atom NEGATE and passes control to the trigger macro which again matches the character string NEGATE in the input and passes control to the ID syntax macro. This time the syntax macro finds a valid identifier (VAR), matches it, and returns control of the scanning process to the NEGATE macro. The NEGATE macro resumes by matching the semicolon in its pattern with one that it finds in the input.

With the successful completion of matching, all text scanned under control of the NEGATE macro and macros called from it is removed from the input and the second statement of the NEGATE macro is executed. The keyword RESCAN indicates that the text produced from the following template is to be returned to the input stream to replace the text that the pattern matched. ID appears in the second statement as a string variable representing the text string argument returned to the NEGATE macro as a result if its first statement's call to the syntax macro ID. Thus the string variable ID has the value VAR. The replacement text is then the concatenation of the strings ID, '=-', ID, and ';' so that the result of the appearance of (2.17) in the input is the production in the output of the statements

```
CALL FCN(NEGATE,VAR);             (2.19)
VAR=-VAR;
```

The next example illustrates how a simplified arithmetic expression can be parsed using syntax macros and converted to another form. A trigger macro is used to find the expressions to be converted.

Arithmetic expressions having the infix operators add (+) and multiply (*) must be converted to reverse polish notation in which each operand and operator is separated from the others by a comma. The precedence of the multiply operator in the original expression is higher than that of the add operator, and parentheses can be used in the normal way to change the order of evaluation. The expressions to be converted will appear in a statement preceded by the identifier POLISH and terminated by a semicolon. This statement is replaced by a statement containing the polish expression. The following trigger macro is used to find the expressions.

```
MACRO TRIGGER: 'POLISH' EXPRESSION ';';
   RESCAN EXPRESSION ';';                (2.20)
   END MACRO;
```

The syntax macro EXPRESSION is called to recognize the original expression, and must return the converted expression to the POLISH trigger macro. This macro and two others it must call are defined

```
MACRO SYNTAX: EXPRESSION
       TERM '+' EXPRESSION | TERM2:TERM ;
       IF TERM2 THEN
           ANSWER TERM2;
       ELSE
           ANSWER TERM ',' EXPRESSION ',+' ;
       END IF
       END MACRO;

MACRO SYNTAX: TERM
       FACTOR '*' TERM | FACT2:FACTOR ;
       IF FACT2 THEN
           ANSWER FACT2;                      (2.21)
       ELSE
           ANSWER FACTOR ',' TERM ',*' ;
       END IF
       END MACRO;

MACRO SYNTAX: FACTOR
       '(' EXPRESSION ')' | ID | NUM ;
       IF EXPRESSION THEN
           ANSWER EXPRESSION ;
       ELSE
           ANSWER SOURCE;
       END IF
       END MACRO;
```

A number of new items are introduced in the definition of the EXPRESSION syntax macro. The pattern for EXPRESSION is really two patterns which are separated by an alternation symbol (|). If the first pattern fails to match the input the second will be tried. The EXPRESSION macro will fail if neither of these two sub-patterns can match the input. The pattern of the EXPRESSION macro has two calls to the TERM syntax macro. In order to distinguish between these it is necessary to give one of them the label TERM2. The text returned to the EXPRESSION macro by its second call to TERM can then be referred to in later statements by the string variable TERM2. If a syntax macro call is not labelled a default label the same as the name of the macro is assumed. Note that the pattern for the EXPRESSION macro calls itself recursively. The second to the last statements of the EXPRESSION macro definition form what is now called the replacement procedure. The first of these statements tests for the existence of the TERM2 string variable. If the second half of the pattern matched the input the TERM2 variable will exist and the statement immediately following the IF clause will be executed. Otherwise the first half of the pattern must have matched successfully and the ANSWER statement following the ELSE clause is executed. The keyword RESCAN has been replaced by ANSWER, which indicates that the text produced by the following template is to be returned to the calling macro as an argument rather than to the input stream. In the third definition a call to the NUM syntax macro appears. NUM, like ID, is assumed to have been defined and to have a pattern which matches a contiguous sequence of digits. The text matched by NUM is returned unaltered to the calling macro. The keyword SOURCE is introduced in an ANSWER statement of the FACTOR macro. SOURCE is a string variable which always equals the text matched by the entire pattern of the macro. Its use avoids the need for two ANSWER statements - one to execute if the call to ID was successful, and another for the call to NUM.

If the expression macro EXPRESSION is activated with the text A*B in the input, it will activate the TERM macro, which will activate the FACTOR macro, which will search for the character "(". Upon failing, FACTOR will try its next alternative and call the ID primitive which would successfully match the A and return it to TERM. TERM will then match the "*" character and then call TERM, and so on. Eventually the first half of the EXPRESSION pattern will search for a "+" and fail, causing this entire process to be redone, this time successfully, using the second alternative in the EXPRESSION pattern.

The conversion to polish notation is quite easy. Each execution of each replacement procedure assumes that the syntax macros called by it return polish expressions. If necessary, the replacement procedure will move an infix operator from between to behind the two polish expressions it receives, thus making the entire expression that it returns polish also. If the first alternative of the pattern of the FACTOR macro matches the input, the parentheses, which can be thought of as surrounding a polish expression, are simply removed. The statements

```
POLISH  A+B;
POLISH  A+B*C;                 (2.22)
POLISH  U+V*(X+Y)+Z;
```

will then appear in the output as

```
A,B,+;
A,B,C,*,+;                     (2.23)
U,V,X,Y,+,*,+,Z,+;
```

As an exercise, the reader may wish to trace the conversion of the first two statements in (2.22).

2.3 Top Down Compiling

The syntax macros in (2.21) can be called recursively, although only right recursion is used in their definitions. Because most expressions in higher level languages associate to the left, it is difficult to use a right recursive grammar for their definition. In order to avoid this problem one might think of using left recursion in the definition of the EXPRESSION macro and write

```
MACRO SYNTAX: EXPRESSION
    EXPRESSION '+' TERM | TERM ;       (2.24)
    etc.
```

The first thing that EXPRESSION will do upon being called is to call EXPRESSION, which in turn will immediately call EXPRESSION, and so on until the processor's stacks fill causing a diagnostic to be printed and the first alternative of the original call to EXPRESSION to fail (except for this action, processing would then continue unaffected). This problem is typical of the construction of a syntax recognizer for a recursive descent compiler, and is normally eliminated by the introduction of iterative notation for syntax definition. (See 13.2, "References" on page 74, item 2) A simple example of an iterative pattern can be written

```
<3, ID>;                       (2.25)
```

This pattern will match any sequence of three or more identifiers which must be separated by blanks. Thus the pattern

```
'BEGIN' <3, ID> ';'            (2.26)
```

will match each of the first two lines of (2.27), but not the third.

```
            BEGIN A BB CC;
            BEGIN FORTRAN ALGOL PL1  THIS  THAT  OTHER;    (2.27)
            BEGIN THIS THAT;
```

The pattern of the EXPRESSION macro in (2.21) can then be replaced by

```
                     <1, TERM / '+'>;                      (2.28)
```

The integer appearing just inside the left bracket indicates that at least one TERM must be found in the input in order for the whole pattern to be considered successful. The part of the pattern occurring after the slash is considered a separator or delimiter which must appear between successive appearances of the portions of the input matched by the TERM macro. Control of the matching process can be thought of as passing out of the iterative construct at the slash, or exit symbol, rather than at the right hand end. Thus the sequence of three TERMs and two separators

```
                        A + B*C + D                        (2.29)
```

will be matched by the pattern (2.28), while the same pattern except for the slash would only match the sequence

```
                         A + B*C +                         (2.30)
```

The string variable TERM in the replacement procedure for (2.28) must now become an array of strings, so that TERM(2) will represent the text returned by the second matching of the TERM syntax macro. The pattern of the TERM macro can also be rewritten using iteration, so that the minus sign can now be safely added to the EXPRESSION macro and the division operator to TERM without any undesirable effects. The iterative pattern is useful for many other purposes. For example, if the syntax macro STATEMENT has been written which will match any statement in a given language, a block of statements might be matched by the pattern

```
                'BEGIN' <1, STATEMENT> 'END' ;             (2.31)
```

If the syntax macro patterns can be called syntax recognizers and their accompanying replacement procedures called semantic routines, the STEP processor becomes a top-down recursive descent compiler generator. One trigger macro (whose trigger could be, for example, SUBROUTINE) would be needed to start the compilation process, which could thereafter proceed through calls to syntax macros. The example in appendix C shows how a FORTRAN preprocessor was implemented in this way. The "master" macro in this example is triggered by the appearance in the input of the string

```
                   BEGIN PROCESSOR SCAN                    (2.32)
```

and then passes control of the scan to the pattern

```
   <0, SUBROUTINE | FUNCTION | BLOCK_DATA | MAIN_PROGRAM >  (2.33)
```

The above pattern will attempt to find zero or more occurrences of a SUBROUTINE, FUNCTION, or BLOCK DATA subprogram or a main program in the text that it scans. The syntax macros for SUBROUTINE, etc. must next be defined, then the syntax macros that they will call are defined, and so on, so that the definition of the language proceeds from the topmost general constructs down to the more detailed parts such as the syntax for statements, then variables, operators, etc. The various facilities needed by a compiler, such as stacks, symbol tables, etc. have not yet been mentioned but are implemented in the STEP processor. When operating as a compiler generator, use is often made of the OUTPUT statement, which is similar to ANSWER and RESCAN except that it causes text to be passed directly to the output file.

A syntax macro might then be called to recognize a term in an arithmetic expression, generate assembly code to calculate the term and store it in a temporary variable. The assembly code would then be written to the output file while the name of the temporary variable containing the result is returned to the calling macro via the ANSWER statement.

2.4 Recursive Matching

Because of the recursive nature of the STEP processor, it is possible for the input text to activate a trigger macro at any time, even during the matching of another trigger or syntax macro. To illustrate this recursive matching the following macro is defined.

```
          MACRO TRIGGER:
                'SUM OF' A:FORTEXP 'AND' B:FORTEXP ;       (2.34)
                RESCAN '(' A '+' B ')' ;
                END MACRO;
```

where FORTEXP is assumed to be defined as a syntax macro which will recognize any valid FORTRAN arithmetic expression and return it unchanged to the caller. The input

```
                   X = SUM OF VAR AND INT*C ;              (2.35)
```

will then appear in the output as

```
                        X = (VAR+INT*C);                   (2.36)
```

If instead the user writes

```
                X = SUM OF SUM OF X AND Y AND C ;          (2.37)
                    |      |             |
                    1      2             3
```

the SUM macro will be called when the input scan pointer reaches position 1. The SUM macro then controls the progress of the scan until the scan pointer reaches position 2, where, just before the call to FORTEXP, the status of the current matching process for the SUM macro is saved and the SUM macro is reinvoked. When the scan pointer reaches position 3 in (2.37) the second invocation of the SUM macro will have terminated with the input text and scan pointer position shown below

```
                    X = SUM OF (X+Y) AND C ;               (2.38)
                               |
```

The original invocation of the SUM macro now resumes where it left off by calling FORTEXP, which will match the string (X+Y). The statement finally appearing in the output will be

```
                         X = ((X+Y)+C);                    (2.39)
```

Note that (2.34) is something like a converse of (2.21): it translates what are essentially prefix polish notation expressions into infix notation. It is interesting that FORTEXP, which was written to parse expressions in the base language, is effectively called by the first invocation of the SUM macro to recognize an expression (SUM OF X AND Y) of the extended language. To be technically correct, the matching process for the trigger macros should be called reentrant rather than recursive, since a trigger macro cannot call itself: it can only call syntax macros.

The STEP processor allows for the definition of both global and local trigger macros. Whether any macros of either type are active or not, if the input scan pointer reaches the beginning of the occurrence of the pattern of one of the global trigger macros, the status of the current matching process, if any, will be saved and that macro will be activated. The protect option, which is described in section X, allows trigger macro activation to be suppressed by other macros when desired. Local trigger macros are activated in the same way that global trigger macros are except that their definitions must have been nested immediately within the definition of the macro (either trigger or syntax) that is currently active. All syntax macros are globally known.

Although it is difficult to think of using trigger macros to produce a true bottom up syntax recognizer, the matching process for the global trigger macros is certainly bottom up: macros are activated as the input text scan pointer reaches the appropriate strings in the input. The MORTRAN macro processor, which allows only non-recursive global trigger macros, is a good illustration of the power of this type of matching. The use of local trigger macros will allow for more control over the context in which trigger macro activity takes place.

3.0 Overview of the STEP Processor

3.1 Internal Structure

The STEP processor consists of about 1400 lines written in a special implementation language. An ANSI FORTRAN program of about 3500 lines is produced after this implementation language is processed by a set of STEP macros, thus insuring portability of the processor. The user, at his option, may write macros which will translate the languages he implements into ANSI FORTRAN, so that his programs, as well as the processor, are portable.

The processor consists of the following major components:

1. Input reader - input on cards is transformed into input suitable for the main processor.
2. Matcher - performs all pattern matching as well as trigger macro invocation.
3. Executive - an interpreter which executes the macro replacement procedure.
4. Output processor - the code produced by the main processor is written to the code file.
5. Pattern compiler - the pattern portion of a macro definition is translated and checked.
6. Replacement compiler - the replacement procedure portion of a macro definition is compiled and checked.
7. Linker - the compiled macro is linked to contained macros, other trigger macros, and perhaps the syntax macro symbol table.
8. Symbol Table Facility - a collection of subroutines used to allocate, search, and update symbol tables.
9. Miscellaneous - various utility subroutines used to convert integers to text, do garbage collection, initialization, and other odd jobs.

3.2 Basic Functioning of the Processor

When the STEP processor begins execution, the input file is read and reformatted by the input reader and sent to the matcher. The matcher attempts to find trigger macros which will match the input text which starts at the location indicated by the input scan pointer. If none is found, the output processor is called to write out one or more of the characters, after which it updates the scan pointer. This process will continue until a trigger macro matches or the end of the input is reached.

When a trigger macro (or a macro called explicitly or implicitly from it or its descendents) is in control, no more characters are passed to the output processor (unless a type four output statement is executed). Characters are still received from the input reader as the matching process requires them.

For each of these macros which successfully matches the input, a corresponding replacement procedure is activated which normally returns the replacement to the point from which the macro was called. If, as is the case for a trigger macro, the call originates from the input text then the replacement is normally substituted for the text which was matched, the input scan pointer is reset to the beginning of this replacement text, and the matching process resumes. If, as is the case for a syntax macro, the the call originates from the pattern of another macro, the replacement is returned to the calling macro as an argument. Text can also be generated by the replacement procedure which, by means of the OUTPUT statement, is passed directly to the code or an auxiliary file.

When an outer level trigger macro (the one that began to match when no other macros were active) has successfully matched and finally terminates, it normally will have generated text to replace the input text which it (and the macros called from it) matched, and will have reset the input scan pointer to the beginning of this text, after which the process described at the beginning of this subsection resumes.

3.2.1 Initial Conditions

The only macros initially present in the processor are five primitive syntax macros (described in section V) and the MACBOOT trigger macro. The MACBOOT macro will match and pass to the internal compilers any text string beginning with the string MACBOOT followed by a base language STEP macro definition, the format of which is described later. Among the first macros defined using this ″bootstrap″ macro will normally be a more sophisticated macro defining macro. Macro definitions may appear anywhere in the input, but will affect only the input text processed after they are compiled.

3.2.2 Input/Output Files

The five file types presently used by the STEP processor are:

1. All input, including STEP control cards and macro definitions, is read from the source file. This file must include a special control card which terminates the input.
2. The listing file contains a listing of the source file, which will include macro definitions along with any diagnostics which the processor generates. In the left margin of each source line are: (1) a character which will be an apostrophe (') or quotation mark (″) if the beginning of the source line is within a quoted string of either type; and (2) the nest level count, which is described in the next section. If appropriate flags are set by macros or control cards, special macro definition listings and macro execution traces will also appear in the listing file.
3. All text produced by WARN statements in macro replacement procedures is written on the warning file. Normally this and the listing file are merged (same unit numbers used for each).
4. The code file will contain the ″object code″ produced by macro translation of the source.
5. There may be up to six auxiliary files. Text may be written to these by means of an OUTPUT statement. These files may then be rewound and copied into the code file in any order desired through the use of the MERGE statement. The auxiliary files are written in an internal format and so may only be used as ″scratch″ files by the processor.

4.0 Input Reader

4.1 General Input Formatting

The input reader serves as the interface between the text in columns 1 to 72 on input cards, and the continuous stream of characters received by the pattern matcher. The current input reader allows comments to be enclosed in quotation marks (″). Comments are changed to a single blank by the input reader and so are never seen by the matcher. By including the appropriate control card in the input the user can cause the input reader to supply the closing quotation mark for those comments not otherwise closed by the end of a card.

Except for those contained in quoted strings, all multiple contiguous blanks, including those resulting from the conversion of a comment, are changed to a single blank. Blanks are compressed in order to save space in text storage areas and to allow the processor to run a little more efficiently. Multiple blanks may still appear in the input as a result of input text replacement by trigger macros.

The nest level count, mentioned in the previous section, is simply an integer whose value is printed in front of each line of the source listing. This number is normally controlled by the macros in a way to be described later. It may be convenient, however, to use the bracket symbols (`'<'` and `'>'` on most machines), as in the language implemented by the MORTRAN processor, to delimit groups of statements. This is identical to the use of the DO-END group in PL/I. As an aid to the use of this symbol, the nest level count can be updated by the input reader. In this case the nest level is incremented each time a left bracket (`<`) is encountered and decremented for a right bracket. To be counted, a bracket must not be in a comment, in a quoted string, or nested within parentheses. It is also possible to allow the nest level count to control the indentation of the source listing. Instructions on how to control the nest level count are given in the description of the GLOBAL array in 7.0, “Macro Replacement Procedure Definition: Introduction, Data Types, and Expressions” on page 37.

Control cards are written with `'/C'` in the first two columns. The entire control card is intercepted before it reaches the input reader, and so is not passed on to the matcher. The input to the processor must always be terminated by `/CE` in the first three columns. The only other type of control card currently allowed is of the form

```
/C{INTEGER},{INTEGER}
```

where the second integer may be preceded by a minus sign. The purpose of this control card is given in the description of the `GLOBAL` array in 7.0, “Macro Replacement Procedure Definition: Introduction, Data Types, and Expressions” on page 37.

4.2 Processor Initialization

The first five cards encountered by the input reader contain special initialization information used by the processor and are here shown in the form assumed by this manual.

```
  62   9  10  37
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_$ ,.+-*/()=;:'"#@?|&¬<>!%
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_$ ,.+-*/()=;:'"#@?|&¬<>!%''
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_$ ,.+-*/()=;:'"#@?|&¬<>!%''
   6   9   6   1   2   3   4   7   8
                                                         (4.1)
```

The first four of these cards are discussed in the next subsection. The fifth card contains up to nine numbers in 9I4 format which are used to set the FORTRAN unit numbers for the listing, code, warning, and up to six auxiliary files respectively. The unit number for the initial source file cannot be set in this way and so must be set at processor generation time. The processor as initially received is a FORTRAN program for which the first routine is a BLOCK DATA subroutine. The last statement of this routine is a DATA statement in which the variable IIU, which is the unit number for the initial source file, is initialized to 5. This is easily changed if necessary.

4.3 Character Set

Characters are divided into four major classes in the STEP processor. These are (1) the digits; (2) the letters, which depending upon the initialization, may contain a few additional characters; (3) special characters used by the processor, such as "|", "<", and ">", which are used in macro pattern definition; and (4) all other characters. Characters in classes three and four are also called delimiters.

Each character that is to be accepted as input to the processor is mapped into an integer ranging from zero up to some maximum number M. If X is the maximum positive integer allowed in the type of storage locations chosen to hold the mapped characters, then the relationship between M and X can be expressed

```
              M <= (X-3)/2                    (4.2)
```

The input character set and its internal mapping are determined by the first few cards read when the processor is started. The first of these contains four numbers in 4I4 format, as shown in the first line of (4.1). Let these numbers be represented by A, B, C, and D. A then is the total number of distinct input characters. The internal numbers representing the characters will then range from zero to $A-1$. The numbers representing the digits will range from zero to B, which will usually equal nine. If, for example, B is equal to seven, all integer arithmetic and conversion to text done by the processor will be in octal. The lowest number representing a letter will be C and the highest D. The twenty four characters (including blank) after the dollar sign ($) are required for use by the processor (actually "#", "%", "!", and "?" are not used for anything at present). The ordering of these characters or their substitutes is important, but they are in no way reserved from any other possible use. The following relationships must hold:

```
         B>=C-1    C<=D    D+24<A              (4.4)
```

These twenty four characters are called special characters as mentioned above. More characters can easily be added to the character set beyond these special characters if desired, as long as the value of A is increased accordingly.

The characters making up the input set are read starting from the second card in the input. If more than 64 characters will be used then multiple cards must be used with characters starting in the first column and 64 characters on all but the last. The positions of the characters on the input card(s) correspond to their position in the internal map with the character in the first column of the first card being mapped into zero. In (4.1) the input set consists of 62 characters and is given by the second line. Note that the characters '$' and '_' (internal values 36 and 37) are considered to be letters.

SPECIAL NOTE: In the current processor the four numbers of the first initialization line are fixed to the values given in (4.1) until further modifications are completed.

Note that while characters can be switched and substituted on this card, the results can be confusing if it is not done with care. For example, the coding and actions of arithmetic expressions in macro definitions might be difficult to correlate if the positions of the characters ″+″ and ″*″ were exchanged. If ″!″ were mapped into the number D+16 (D=37 for (4.1) so this is 53 in the internal map and column 54 on the input card) in place of ″|″, the user would have to remember to separate alternative portions of a macro pattern by ″!″ and that ″!″ would then be used to denote the ″OR″ operation in the replacement procedure base language.

Each of the characters appearing on the input mapping cards must be distinct. This includes the character blank. Any character not on this initialization card(s) that appears later in the input will produce a diagnostic and be ignored.

Following the cards describing the input character set come those for the output set. The internal character set is mapped into the characters given by the appropriate columns on the output card(s) before being written. These cards follow the same formatting rules as do the preceding cards and make use of two additional columns which will be described in the next subsection. Except for these last two, the columns of this card(s) are normally identical to the corresponding columns of its predecessor(s), but this is not necessary. For example, to gain the flexibility of having two characters, ″.″ and ″!″, represent ″.″ the user could change column 61 on the output map card in (4.1) to ″.″. The two characters would then be treated differently by the processor until they are written into the code file, where they would both appear as ″.″. The output map is also used to transform the internal characters for all other forms of output except the listing file. These will include traces of the matching process and listings of the macro compilers. The characters in the output map need not be distinct.

The fourth line in (4.1) is a second output character set which is used to map any text produced with the NOTRIG option (see the RESCAN statement in 8.0, “Macro Replacement Procedure Definition: Statements” on page 46) that appears in output produced by the trace facility. The first output character set is used when NOTRIG text is written by the output processor. In (4.1) the second output character set is the same as the first, but it need not be. The letters in the fourth line of (4.1) could be lower case, for example.

4.4 Quoted String Conversion

A quoted string consists of a string of characters delimited by apostrophes ('). The characters within a quoted string may include quotation marks or multiple contiguous blanks, all of which will be passed on to the matcher.

The manipulation of quoted strings is complicated by the fact that these strings will often contain an apostrophe, which is the same character as that used to delimit a quoted string. The usual method of inserting an apostrophe into a quoted string is to code it as two adjacent apostrophes. Thus, in order to form a quoted string from the characters

```
                    DON'T                           (4.5)
```

the user would code

```
                  'DON''T'                          (4.6)
```

The four apostrophes in the above quoted string are normally interpreted differently by the user. The first and last apostrophes are string delimiters, while the other two represent the single character apostrophe.

In order to make character strings easier to handle, the input reader changes two adjacent apostrophes that are contained by a quoted string into a single apostrophe. The apostrophes used to delimit the quoted string are changed into internal string delimiter characters. After this is done, the string (4.6) will contain five rather than six characters and the substring and string length functions (to be introduced later) will work as the user intends. If the internal string delimiter character is represented by a lower case d and the characters to the left of each example are not in a quoted string then the following examples illustrate the transformation made by the input reader.

```
'DON''T'            ->      dDON'Td
'''STRING'''        ->      d'STRING'd           (4.7)
X='STRING:''ABC'''  ->      X=dSTRING:'ABC'd
TRIGGER: 'X=''STRING:''''ABC''''''';    ->
                      TRIGGER: dX='STRING:''ABC'''d;
```

In addition to having the quoted string contain the number of characters intended, the conversion of quoted strings will make the matching of quoted strings simpler: after matching the opening string delimiter the matching process proceeds until the closing string delimiter is found. Since they are a different character, any apostrophes that might be contained by the string cannot stop the matching process prematurely. Thus the macro

```
            MACRO TRIGGER:
                '''' BAL '''' ;                (4.8)
                etc.
```

will match an entire quoted string including the string delimiters, even if the string contains apostrophes. It is perhaps confusing at this point to note that the apostrophes which are to match the string delimiters must be doubled since they are contained in a quoted string, but this fact will be clarified shortly.

The last line in (4.7) is actually a pattern to match the previous line in (4.7), but since patterns to match specific characters must be enclosed by apostrophes it was necessary to double all of the apostrophes in the third line to produce the pattern in the fourth. If the string in the macro pattern is to match the string as it would appear in the input, however, its internal representation must be identical to the string in the input. Therefore it is necessary to ″undouble″ the apostrophes in quoted strings appearing in macro definitions twice. For example,

```
     TRIGGER: 'X=''STRING:''''ABC''''''';          (4.9)
```

is converted by the input reader to

```
        TRIGGER: dX='STRING:''ABC'''d;           (4.10)
```

which is converted by the macro compilers to

```
TRIGGER: mX=dSTRING:'ABC'dm;            (4.11)
```

The string delimiter ″d″ is converted by the macro compilers to a pattern string delimiter ″m″. Single apostrophes are converted by the macro compiler to a string delimiter in the same way as they are by the input reader. The string delimited by the pattern string delimiters in (4.11) is now the same as the third string in (4.7) after it is converted by the input reader. This second conversion is done for all quoted strings in a macro definition. Thus if the user codes the replacement procedure statement

```
RESCAN '''DON''''T''';                (4.12)
```

the compiled RESCAN statement would contain the (delimited) text

```
mdDON'Tdm                             (4.13)
```

When executed, this RESCAN statement would place the string

```
dDON'Td                               (4.14)
```

into the input.

Quoted strings which were converted at input must be converted back by the output processor. Thus the quoted string in (4.14), if not further converted by the macros, will appear in the output as

```
'DON''T'                              (4.15)
```

Apostrophes in (4.14) are changed to two adjacent apostrophes and string delimiters are changed to single apostrophes. The output processor is not concerned with the macro pattern string delimiters since these occur only in compiled macro definitions. Other functions of the output processor are described in 9.0, “Macro Replacement Procedure Definition: Intrinsic Functions, Listings, and Object Code” on page 57.

Conversion of quoted strings then occurs in three places in the processor: the input reader, the macro compilers, and the output processor. Text returned to the input via the RESCAN statement does not pass through the input reader since it is necessarily derived from text that has already been processed there.

The last two columns on the output map card(s) contain the characters into which the internal representations of the string delimiter and macro pattern string delimiter characters will be converted on output. Since these characters are internally generated they will never appear in the input and so do not appear on the input map card(s). Although these two characters are never written into the code file, they can appear in the listing file as a result of the trace option, which causes the text produced by macro replacement procedures to be listed. An apostrophe is normally used to represent each of these characters, as shown in (4.1). If extra information is needed from the trace output, however, these two internal characters can be mapped into a different external representation, such as a lower case ″d″ and ″m″.

Normally it is not necessary to give any thought to the conversion of quoted strings that occurs within the processor. All that the user normally needs to remember is that if an apostrophe is needed, normally an apostrophe should be coded. If the apostrophe must be in a quoted string it should be coded as two adjacent apostrophes. If an apostrophe must be placed into a quoted string that is contained by another quoted string it must be coded as four adjacent apostrophes, etc. This description has been included for those few occasions when the user will need to know exactly what is happening to quoted strings during the matching process.

4.5 Identifiers

All identifiers used by the STEP system must begin with a letter (see the above character set subsection) and contain only digits and letters. Embedded blanks are not allowed. Identifiers are used as syntax macro names and labels in the patterns, and as statement labels, variable names, and function names in the replacement procedures. The `ID` primitive macro, which is described later, will match any identifier.

4.6 Text Atomization

The input processors of many compilers contain lexical analyzers which transform the stream of input characters into a stream of tokens, or atoms, which would be identifiers, integers, perhaps floating point numbers, and so on. These functions are not performed by the STEP input reader, so that the pattern matcher will deal with the input on a character by character basis. Because of the way in which the scan is controlled both by the processor when no macros are active (see 9.0, “Macro Replacement Procedure Definition: Intrinsic Functions, Listings, and Object Code” on page 57) and by the macros themselves (see 6.0, “Macro Pattern Definition: Pattern Construction” on page 29) an effective form of text atomization usually exists. For this reason an atom is defined to be any contiguous sequence of alphanumeric characters, any contiguous sequence of blanks, a quoted string, or any non-alphanumeric non-blank character.

5.0 Macro Pattern Definition: Match Items

The syntax used for the definition of macro patterns has been to a great extent borrowed from that used by the IBM Language Point 2257. (See 13.2, “References” on page 74, item 4) It bears some resemblance to BNF notation, with the significant addition of iterative notation. The necessity of using right recursion in the syntax definitions for top-down parsing is thereby avoided.

The only items within a pattern that will cause matching to take place are calls to syntax macros (which are written as identifiers) and quoted strings. These two constructs, calls to syntax macros and quoted strings, will be called match items. All other syntactic symbols in the pattern control the way in which matching occurs, and are therefore called control symbols. Example (2.14), which is reproduced here, is a pattern which contains four quoted strings and three syntax macro calls.

```
        'ADD' A:BAL 'TO' B:BAL 'AND STORE INTO' ID ';'      (5.1)
```

A macro pattern is then defined as a collection of match items together with control symbols which appears as the first statement of a macro definition. Later this definition will be modified with the introduction in 8.0, “Macro Replacement Procedure Definition: Statements” on page 46) of the `SCAN` statement, which allows macro patterns to appear in the replacement procedure.

5.1 Pattern Strings

A quoted string within a pattern is called a pattern string, and is matched on a character by character basis with the input text. Exceptions to this character by character matching occur for blanks in the input. One blank in a pattern string will match one or more contiguous blanks in the input. One or more blanks in the input for which no corresponding blanks in the pattern string exist are allowed if

1. the blanks occur before the character in the input which matches the first character in the pattern string.
2. the blanks are delimited on at least one side by a non-alphanumeric character. Consecutive blanks in pattern strings are compressed to a single blank when the patterns are compiled.

For example, the pattern string `'ABC(123)'` will match the input strings

```
            ABC(123)                 ABC  (  123  )         (5.2)
```

but will not match

```
            AB C(123)                ABC ( 1 2 3 )          (5.3)
```

The pattern string `'GO TO 123'` will match the input strings

```
            GO TO 123                GO   TO   123          (5.4)
```

but will not match

```
            GOTO 123                 GO  TO123              (5.5)
```

The pattern string `'HERE      TO      THERE'` will match the input strings

```
            HERE TO THERE            HERE    TO    THERE   (5.6)
```

In order that pattern string matching terminate at what will normally be an atom boundary in the input an otherwise successful match is made to fail if the last input character matched is alphanumeric and the character immediately following it is also alphanumeric. This requirement may be relaxed by coding an asterisk immediately after a pattern string when a macro is being defined. Thus the macro

```
        MACRO TRIGGER: 'PI'; RESCAN '3.14'; END MACRO;      (5.7)
```

will convert the input

```
                    PI SPIN  PIN  (PI+PIT)*A;               (5.8)
```

into

```
                  3.14 SPIN  PIN  (3.14+PIT)*A;             (5.9)
```

The same input, if processed by the macro

```
       MACRO TRIGGER: 'PI'*; RESCAN '3.14'; END MACRO;     (5.10)
```

would become

```
               3.14 SPIN  3.14N  (3.14+3.14T)*A            (5.11)
```

The PI in SPIN will not be converted unless the input scan pointer is left pointing to the P in SPIN by some other macro whose pattern could, for example, end with 'S'*. Normally this will not be the case.

Pattern strings of zero length are not allowed and will cause the processor to generate a diagnostic message if encountered in a macro definition.

Exceptions to some of the rules given above for pattern string matching occur when quoted strings or portions of quoted strings are being matched. Those portions of a pattern string occurring after an odd number of apostrophes (each of which was originally coded in the pattern string as two adjacent apostrophes) are treated somewhat differently during macro definition in that multiple contiguous blanks are not compressed. When these portions of a pattern string are being matched against the input, blanks are treated as any other character (e.g. three blanks in the pattern string are required to match three blanks in the input). These rules would then apply when the pattern string begins matching outside a quoted string in the input and then enters the string by matching its left string delimiter. Normal matching conditions would again prevail if the pattern string matched the quoted string's closing delimiter.

A pattern string can be considered to simply return to the surrounding pattern an unaltered copy of the text that it matches.

5.2 Syntax Macro Calls

A syntax macro call will itself match no text unless it is a call to one of the predefined STEP primitive syntax macros. Otherwise control will be passed to the pattern of the syntax macro that is called. Eventually, of course, control must be passed to either a pattern string or a primitive syntax macro.

5.2.1 Primitive Syntax Macros

Primitive syntax macros have been implemented primarily for reasons of efficiency. While each of them could be replaced by a syntax macro using only pattern string match items, the patterns of such macros would have long alternative sequences of single character quoted strings that would take considerable amounts of time when matching. As an example, the ID primitive macro could be replaced by the following three syntax macros.

```
MACRO SYNTAX: ID (P=2)
        LETTER <0, ¬<' '> <LETTER | DIGIT >> ;
        ANSWER SOURCE;
        END MACRO;

MACRO SYNTAX: LETTER
        'A'* | 'B'* | 'C'* | 'D'* | 'E'* | 'F'* |  (5.12)
        'G'* | 'H'* | 'I'* | 'J'* | 'K'* | 'L'* |
        'M'* | 'N'* | 'O'* | 'P'* | 'Q'* | 'R'* |
        'S'* | 'T'* | 'U'* | 'V'* | 'W'* | 'X'* |
        'Y'* | 'Z'* | '_'* | '$'* ;
        ANSWER SOURCE;
        END MACRO;

MACRO SYNTAX: DIGIT
        '0'* | '1'* | '2'* | '3'* | '4'* | '5'* |
        '6'* | '7'* | '8'* | '9'* ;
        ANSWER SOURCE;
        END MACRO;
```

The reader should not concern himself with completely understanding the first of these three example macros at this point, since it makes use of some as yet undefined constructs. The pattern for the ID macro defined in (5.12) must first match a letter, after which it may match zero or more letters or digits which may not be preceded by a blank.

Currently five STEP primitive macros are available, but more can easily be added if required.

ID: The ID macro will match any identifier, which is defined as a string of one or more letters and digits beginning with a letter and containing no embedded blanks. The character after the identifier will be a delimiter. Note that there is no requirement on the character immediately before the identifier. In other words, when the ID macro begins to scan the input to look for an identifier it starts at the current location of the input scan pointer and will not consider any characters which occupy preceding locations in the input. Thus if the input contains the string

```
                    ABC001                    (5.13)
                      |
```

with the input scan pointer at the position shown, the ID macro will successfully match and return the identifier C001. Note that no match item that can be defined in a STEP macro can test whether or not it begins to match on an atom boundary, although most match items do test that the matching process completes on one. Normally this should not be a problem, because if the previous matching process terminates with the input scan pointer on an atom boundary, as it normally will, the next matching process must then begin on an atom boundary.

NUM: The NUM macro will match any string of one or more decimal digits. Matching terminates when a character other than a digit is encountered. Again, there is no requirement on the character preceding the matched string.

STR: The STR macro will match any quoted string. It will match the entire string including the delimiting apostrophes, even if the string contains an embedded apostrophe (which must have been coded as two adjacent apostrophes).

DEL: The DEL macro will match any single non-alphanumeric character. Multiple blanks are matched as one whether they occur inside of a quoted string or not.

CHAR: The CHAR macro will match any single character. Multiple blanks are matched as one whether they occur inside of a quoted string or not.

Except for CHAR and DEL, all of the above primitive macros (and also the pattern strings) will additionally ″match″ any leading blanks that might occur as they begin to scan the input. Trailing blanks are not matched, since a call to DEL or CHAR, or a pattern string expecting a blank, could occur in the pattern immediately following one of the calls to a primitive syntax macro.

BAL: The BAL syntax macro, or formal argument as it was called in 2.0, “Macro Processing” on page 4, requires either a pattern string or an alternative sequence of pattern strings to immediately follow it. For the present this discussion will assume that BAL is followed by a single pattern string. When a BAL syntax macro is called during the matching of a pattern string against the input, it will match any string (possibly of zero length) up to the first occurrence of the input of a string matching its following pattern string. BAL returns the string that it matches unchanged. The following examples illustrate the use of BAL.

```
PATTERN         INPUT TEXT             FIRST       SECOND
                                        BAL          BAL
                                      RETURNS      RETURNS

'ABC(' BAL ')'  ABC(123)                123                 (5.14)
'ABC'BAL'123'   ABC DEF 123             DEF
'ABC'BAL'%23'   ABC%23                  zero length
BAL','BAL';'    A,B,C,D;                A           B,C,D
```

BAL will not match a string containing a semicolon, unbalanced parentheses, or part of a quoted string. It may match text containing an entire quoted string, however. The following examples illustrate these restrictions.

```
PATTERN           INPUT TEXT            FIRST       SECOND
                                         BAL          BAL
                                       RETURNS      RETURNS

BAL','BAL';'      A(4,6 B(3);           *** NO MATCH ***  (5.15)
BAL','BAL';'      A(4,6),B(3);          A(4,6)      B(3)
'AB'BAL'XYZ'      ABC;XYZ               *** NO MATCH ***
'AB'BAL'XYZ'      ABC'XYZ'D XYZ         'XYZ'D
'AB'BAL'XYZ'      AB)(XYZ               *** NO MATCH ***
```

An exception to the above rules arises when the text matched by BAL contains a quoted string. This text may then contain unbalanced parentheses and semicolons if these characters are also contained by the quoted string. For example, the pattern `'AB'* BAL 'XYZ'` will match the input `ABC'));('XYZ`. In this case `BAL` will return the text `C'));('`.

In addition to the restrictions mentioned above, the point in the input at which the `BAL` ceases and its trailing pattern string begins to match must be an atom boundary. In other words, at least one of the characters adjacent to this point must be a delimiter. For example, if the pattern `BAL 'ABC'` is used to match the input `123ABC*ABC`, the `BAL` syntax macro will match the text `123ABC*` and its trailing pattern string will match the second occurrence of `ABC`. A quoted string is considered for this purpose to be an atom, so that, as mentioned above, `BAL` is not allowed to match across the left boundary of a quoted string without matching the entire quoted string. In the fourth example of (5.15) the first `XYZ` is in a quoted string, and if the pattern string trailing `BAL` matched it, `BAL` itself would have matched only the left quote of the quoted string `'XYZ'`.

It is also possible for `BAL` to be followed by an alternative sequence of pattern strings which must appear within brackets. `BAL` will then match all text up to the first occurrence in the input of text matching any one of the alternative pattern strings that follow it, provided, of course, that the above described restrictions on unbalanced parentheses, semicolons, and atom boundaries are satisfied. For example, the pattern `BAL<'STOP'|';'>` will match the text `"ONE TWO STOP;"` except for the semicolon and `BAL` itself will return the text `"ONE TWO"`. If the same input were to be matched by the pattern

```
BAL<A:'STOP'|B:'STO'*>                    (5.16)
```

`BAL` would return the same text and the pattern string labelled `A` would match the identifier `STOP`, since the leftmost pattern strings in the alternative sequence are tried first. If pattern (5.16) were used to match the input `"ONE TWO STOPPED"` then `BAL` would again return the same text as before and this time the pattern string labelled `B` would match the first three characters in `STOPPED` since it is not constrained by the lack of an atom boundary as is the first pattern string in the sequence.

ATOMS: The `ATOMS` syntax macro behaves exactly as does `BAL`, except that it is allowed to match strings containing semicolons and unbalanced parentheses. Because of this more care must be taken when using `ATOMS` than when using `BAL`, since there is nothing to stop `ATOMS` from matching the entire input stream if the appropriate terminating string is not found there. `ATOMS` will fail only if it attempts to match beyond the end of input.

`ATOMS` can be used to match a portion of the input that was incorrectly coded. For example, the macro

```
MACRO SYNTAX: STATEMENT
     DO_STATE    |
     IF_STATE    |
     ASSIGN_STATE|
        .
        .
        .
     WRITE_STATE |                           (5.17)
     ATOMS ';'    ;

     IF ¬ATOMS THEN
        ANSWER MATCH;
     ELSE
        WARN '**ERROR-' ATOMS '-DELETED';
     END IF
     END MACRO;
```

will always match either a statement of whatever language is defined by the syntax macros that it calls, or will match an arbitrary string of text up to and including a semicolon and declare it to be an incorrectly coded statement. Using the STATEMENT macro a pattern to match a subroutine having no arguments can now be written

```
MACRO SYNTAX: SUBROUTINE
     'SUBROUTINE;'
     BODY: <0, ¬<'END;'> STATEMENT>          (5.18)
     'END;' ;
     ANSWER 'SUBROUTINE;' BODY 'END;';
     END MACRO;
```

The negation symbol (¬), which is explained in the next section, will cause the subpattern in loop brackets to fail if "END;" is encountered in the input. It is assumed that any conversion of the input text that might be done is performed by the macros DO_STATE, IF_STATE, etc. which are called by STATEMENT. The above macro will match the subroutine declaration and all statements, correct or incorrect, up to and including the END statement. Incorrect statements are deleted from the ANSWERed text and cause diagnostic messages to be printed.

TEXT: The TEXT macro is similar to the BAL and ATOMS syntax macros except that there are no restrictions of any kind on the text that it will match. The TEXT macro will match the input on a character by character basis up to the first occurrence of the specified string. The text that it returns will contain all of the matched input and may include unbalanced parentheses, quotes and any number of semicolons. Like ATOMS, TEXT will fail only if it attempts to match beyond the end of input. The pattern string (or strings if an alternative sequence is used) that follows TEXT is treated differently than a normal pattern string in that multiple blanks are always retained. At match time this pattern string(s) must match character by character with the input. TEXT and its following pattern string(s) have no regard for atom boundaries. An asterisk following a pattern string which terminates a TEXT macro will be ignored by the macro compilers, since in any case this type of pattern string need not terminate matching on an atom boundary.

An example of the use of TEXT is the pattern TEXT'E'TEXT'IJ', which will match all of the text "ABC'DEF'GHIJ", with the first call to TEXT returning ABC'D and the second call returning F'GH. The pattern

```
                    TEXT<A:''''|B:''''''>                    (5.19)
```

could match text that begins within a quoted string, and would terminate after matching either the closing string delimiter (A) or an apostrophe coded within the quoted string (B). If this is confusing one should reread the portion of section IV dealing with quoted string conversion. Note that TEXT and ATOMS are interchangeable as far as the matching process of (5.19) is concerned.

Note that each of the BAL, ATOMS, and TEXT syntax macros must always be immediately followed either by a pattern string as defined at the beginning of 5.0, "Macro Pattern Definition: Match Items" on page 22, or by a sequence of pattern strings which are separated from each other by alternation symbols. If the latter is the case the entire sequence must be enclosed in angle brackets ("<" and ">"). It is permissible to label a TEXT, BAL, or ATOMS syntax macro call, and is also legal to label either the single pattern string following it or any of the pattern strings in an alternative sequence. The entire bracketed sequence of pattern strings following one of these syntax macros may not be labelled, however. The following patterns should illustrate these rules.

```
LAB1: BAL LAB2:'STOP'                   LEGAL
A:TEXT <';'|B:'END'|C:'RETURN;'>        LEGAL
ATOMS  LABEL:<'THIS'|'THAT'>            ILLEGAL
```

The exact meaning of these labels will be described in the "Match Variable" portion of 7.0, "Macro Replacement Procedure Definition: Introduction, Data Types, and Expressions" on page 37.

In later versions of STEP, it is intended that the primitive macros can be defined by the user at processor generation time. Thus, for example, if many floating point numbers were present in the input text, a primitive macro might be installed to match them more efficiently.

6.0 Macro Pattern Definition: Pattern Construction

6.1 Simple Patterns

The simplest form of macro pattern is composed of one match item, which would be either a pattern string or syntax macro call. Next in complexity is a pattern composed of a sequence of match items. Such a pattern will successfully match the input only if its first match item can match a portion of the input which begins at the location indicated by the input scan pointer when the pattern was activated, and if each following match item can match a portion of the input which begins immediately after the portion matched by its predecessor. If any one of the match items fails to match the input, the whole pattern will fail. An example of this type of pattern is seen in (5.1), which has three match items.

6.1.1 The Alternation Control Symbol

The alternation control symbol is coded as a vertical bar (|) and is used to divide one pattern into two or more alternative patterns. The whole pattern is considered to have successfully matched the input if any one of its alternative patterns successfully matches. The leftmost member of an alternative sequence is always matched against the input first. If it fails, the input scan pointer backs up to the position it held when the failing member began to match and the next alternative is tried. This process continues until one of the alternative patterns matches or all fail. When a member of an alternative sequence matches the input all remaining members to its right are ignored and the entire sequence is considered to have successfully matched. The FACTOR syntax macro in (2.21) is a good example of a simple alternative sequence in a pattern.

```
MACRO SYNTAX: FACTOR
      '(' EXPRESSION ')' | ID | NUM ;         (6.1)
      etc.
```

The first alternative in the above pattern consists of three match items: a pattern string followed by a syntax macro call which is in turn followed by another pattern string. The next two alternatives each consist of one syntax macro call. Note that the scope of the alternation symbol in (6.1) is delimited by either a boundary of the whole pattern or another alternation symbol. Thus the alternation symbol in the pattern

```
A B C | D E F        (6.2)
```

does not apply to C and D only, but to the patterns A B C and D E F.

6.1.2 The Bracket Control Symbols

A group of match items and control symbols enclosed in brackets is viewed by the rest of the containing pattern, as well as by the rest of this manual unless otherwise stated, as a single match item that as a unit will either succeed or fail to match the appropriate portion of input text. For example, the pattern

```
          A  B  C  D  E  F                (6.3)
```

consists of six syntax macro calls, each of which must successfully match the input if the entire pattern is to be successful. This pattern can be rewritten as

```
        A  B  < C  D  E >  F              (6.4)
```

This new pattern will match a certain portion of the input if and only if (6.3) is capable of matching the same text. While (6.3) consists of six match items, the new pattern consists of four. The six syntax macros are called to match the input in the same sequence as before, and the failure of any one of the six to match will still cause the entire pattern to fail, although in a different way than before. If one of the bracketed syntax macro calls were to fail, its failure will now cause its containing pattern, consisting of the three bracketed syntax macro calls, to fail. The failure of the bracketed match item will then in turn cause the entire pattern to fail.

One of the chief uses of the bracket symbols is to delimit the scope of the alternation symbol. Thus it is possible to rewrite (2.5) as

```
MACRO TRIGGER:
     'ADD' TERM1:<ID | NUM> 'TO' TERM2:<ID | NUM>
            'AND STORE INTO' RESULT:ID. ';' ;         (6.5)
     RESCAN RESULT '=' TERM1 '+' TERM2 ';';
     END MACRO;
```

where the match item `<ID|NUM>` can be thought of as a call to a syntax macro which will match either an identifier or a string of digits. Note that a bracketed match item may be labelled as if it were a single match item (which, as far as the surrounding pattern is concerned, it is). Again, the label is used by the macro's replacement procedure to access and manipulate the text (which in this case will be either an identifier or a string of digits) returned by the match item labeled. If the brackets were absent the above pattern would match successfully any one of the following statements:

```
                      38 TO NAME
                     ADD VARIABLE                    (6.6)
                123 AND STORE INTO ABC;
```

Bracketed expressions in a pattern may be nested to any level desired. Thus the pattern

```
   'ADD' <TERM1:<ID | NUM> 'TO' TERM2:<ID | NUM>
         | RANGE:NUM 'ELEMENTS OF' ARRAY:ID.>         (6.7)
         'AND STORE INTO' RESULT:ID. ';' ;
         etc.
```

allows extended ADD statements to be written in which it is now additionally possible to sum a specified number of elements of an array and store the result. Thus either of the following statements can be matched by (6.7).

```
         ADD IVAR TO 123 AND STORE INTO K;           (6.8)
   ADD 43 ELEMENTS OF ARRAY AND STORE INTO IVAR;
```

While the number of nested brackets in a macro definition is not limited, there is a limit on the amount of nesting allowed during the matching process. The number of macros activated but not yet terminated plus the sum of the number of bracketed expressions entered but not yet exited for each of these calls is at any given instant limited to the value of the parameter `$MATCHSAV`. The default value of this parameter is 100 and can easily be changed at processor generation time. Any macro or bracketed match item which causes this limit to be exceeded will be forced to fail by the processor and a diagnostic message will be printed. Except for this action, however, the normal matching process will continue.

6.1.3 Detailed Description of the Matching Process

Enough of the syntax for writing patterns has now been illustrated to enable the reader to understand the following description of the matching process. Because of the detail involved in the following description, the casual reader may wish to skip to the next subsection describing the bracket control symbols.

If the ADD macro in (6.5) has been defined, the processor, after its scan pointer reaches the first character of the string

```
ADD IVAR TO 123 AND STORE INTO K ;          (6.9)
```

will try to match it against the local trigger macros whose definitions were immediately contained in the definition of the macro that is currently active. If no local macros match, none exist, or no macro is currently active, the global trigger macros are then tried. After perhaps trying and failing to match several other trigger macros (macros defined after the 'ADD' macro are tried first), the processor begins to match the pattern in (6.5) with the text in (6.9).

The string ADD in (6.9) is matched character by character with the first match item (which for a trigger macro must be a quoted string) in (6.5). After the first item (or ″trigger″) of the pattern is matched, and with the input scan pointer positioned immediately after the ADD, the processor looks to see if the ADD macro has any local trigger macros that can be matched, and failing this, looks for global trigger macros that consist of, or start with, the string 'IVAR'. If none are found the ID syntax macro is entered but not activated. At this point the processor checks for any trigger macros local to ID that might be tried, but since ID is one of the primitives there are none, so that the ID macro is finally activated.

The processor will not try to match any other trigger macros until the ID macro completes, successfully or not, its task. After ID has successfully matched IVAR plus all leading blanks, the alternation symbol is encountered in the pattern and all remaining items, including other bracketed expressions, within the brackets are skipped. With the input scan pointer immediately following IVAR, the macros local to the ADD macro and then the global trigger macros are searched for one which will match the text ″TO 123 AND...″. When (hopefully) none are found the pattern string 'TO' will be matched with the corresponding text, plus leading blanks, in the input. With the input scan pointer at the first of the leading blanks before the ″123″ the usual search through the local and global trigger macros will be made, after which the ID syntax macro is activated and fails, returning control to the pattern of the 'ADD' macro. Any pattern items, including bracketed items containing alternation symbols, that might have followed the second occurrence of ID in (6.5) will be skipped over until the alternation symbol is reached. The input scan pointer backs up until it is at the location occupied before the current bracketed expression in the pattern was entered (before the ″123″), and the processor prepares to call the NUM syntax macro. The global trigger macros and those local to the ADD macro are not checked at this point because they already were before ID was called. If NUM were to have any local trigger macros they would be matched against the input, however. This matching process will continue until the semicolon at the end of (6.9) successfully matches the corresponding item in the pattern, after which the replacement procedure of the ADD macro is called.

Normally, the replacement procedure for a trigger macro generates what is called rescan text. This text effectively replaces the text that the macro matched, and the input scan pointer is reset to point at its beginning. The input text is not actually replaced, however, since a match failure may cause the input scan pointer to back up to a location before that at which this trigger macro was activated, upon which the original input should be reinstated.

If at some point the ADD macro had failed to match the input in (6.9), the input scan pointer would have backed up to the position it occupied when the ADD macro was first called, that is, before the ADD in (6.9), and any remaining global trigger macros would be called.

6.1.4 Activation Points

When a macro pattern has control, implicit calls to trigger macros may only take place just before a pattern string match item becomes active or a syntax macro is called. Before a pattern string becomes active, trigger macros local to the containing pattern and then the global trigger macros are tried. The same is done just before a syntax macro call is reached, after which the syntax macro is entered and its local macros are tried. Finally the syntax macro itself is activated. If at any time a trigger macro is successful, it will generate a replacement for the input text that it matched, after which both local and then global trigger macros are retried. Finally when all trigger macros available in the current context are tried and none are successfully matched by the current input text, the processor continues with either the match of the pattern string or the syntax macro call. Note that the pattern string 'AND STORE INTO' in (6.5) is one string, or match item, in the pattern and will be matched as a unit, so that trigger macro activity can take place only when the input scan pointer is positioned before the AND and not when it is before the other two identifiers. This could be altered by writing the pattern in (6.5) as

```
'ADD' TERM1:<ID | NUM> 'TO' TERM2:<ID | NUM>     (6.10)
        'AND' 'STORE' 'INTO' RESULT:ID. ';' ;
```

The locations of the input scan pointer for which trigger macros can be invoked are called activation points. Although the activation points are in the input text, their locations depend upon the macro pattern which currently controls the scan. If no macros are active the activation points are defined by the output processor (see 9.0, “Macro Replacement Procedure Definition: Intrinsic Functions, Listings, and Object Code” on page 57) to be at the beginning of each atom in the input. There are then no activation points within any of the input matched by a single pattern string or any of the primitive syntax macros, including TEXT.

The interaction of syntax and trigger macros described above indicates that language extension could be made into a two stage process. Suppose that the base language being extended is FORTRAN. The first stage could then involve the definition of an extended base language for which a top down parse is conducted by a set of syntax macros. This set could be activated by one trigger macro as described in 2.0, “Macro Processing” on page 4. The extended base could possibly be defined to be a structured FORTRAN such as RATFOR, IFTRAN, or MORTRAN. (See 13.2, “References” on page 74, items 3-5.) Further extensions could then be made by trigger macros. Note that these trigger macros, and any syntax macros which they call, would be invoked for any given portion of the input before the corresponding extended base syntax macros are, and thus should produce code in the extended base, not the base, language. The two stages could then be quite independent, since as far as the trigger macros are concerned, the extended base language is the base language.

6.1.5 The Parentheses Control Symbols

Parentheses are used to delimit optional match items in the pattern. The match items within parentheses will be matched as a unit if they appear in the input text, but need not be there for the entire pattern to match successfully. If EXPR matches arithmetic expressions, then the pattern

```
        'FOR' ID '=' EXPR 'TO' EXPR ('BY' EXPR) 'DO'     (6.11)
```

will match both

```
                 FOR I=A+1 TO A*C BY 3 DO                (6.12)
                    FOR I=A+1 TO A*C DO
```

Note that both the 'BY' and its accompanying expression must be present or both must be absent for the optional match item to match. The pattern

```
       'FOR' ID '=' EXPR 'TO' EXPR ('BY') (EXPR) 'DO'    (6.13)
```

would allow for the (probably undesirable) possibility of one of either the BY or its accompanying expression appearing in the input without the other. The parentheses will also delimit the scope of the alternation control symbol, so that

```
                        (ID|NUM)                         (6.14)
```

could be considered by its containing pattern as a call to a syntax macro which would match an identifier, or an integer, or, if neither of these items appeared at the appropriate place in the input, a null string.

6.1.6 The Negation Control Symbol

The negation (¬) symbol is used in conjunction with the bracket symbols to insure that a given portion of text is not present in that which is currently being matched. The bracketed pattern following a negation symbol will signal a failure to the surrounding pattern if it matches the input successfully. If it fails, however, the input scan pointer will back up to its position previous to the negation symbol being encountered in the pattern, the bracketed item will be considered to have successfully matched, and the matching process will resume with the next item in the surrounding pattern. The pattern

```
            'BEGIN' ¬<'FORTRAN'|'ALGOL'> ID 'END'        (6.15)
```

will match an occurrence in the input of the string 'BEGIN', followed by any identifier except 'FORTRAN' or 'ALGOL', which then must be followed by the string 'END'. Only the first two of the following four strings can then be successfully matched by (6.15).

```
                   BEGIN  PASCAL  END
                   BEGIN  FORTRA  END                    (6.16)
                   BEGIN  FORTRAN  END
                   BEGIN  ALGOL  END
```

A left bracket must follow the negation symbol; a parenthesis or a loop bracket (described below) may not. The definition of ID in (5.12) is a good example of the use of the negation control symbol.

6.1.7 Iterative Matching

Iterative syntax rules and patterns to match multiple occurrences of a given construct in the input text can be written by enclosing the appropriate match items and control symbols within loop brackets. Loop brackets are distinguished by placing after the left bracket symbol (<) an integer, which in turn must be followed by a comma. The integer gives the minimum number of times that the pattern within the loop brackets must successfully match the input before the loop construct as a whole is considered successful. The loop brackets may optionally contain the exit (/) control symbol which specifies the point at which control is passed out of the loop pattern. The pattern

```
'INTEGER' < 2, ID / ',' > ';'            (6.17)
```

matches the text (ignoring normal recursive trigger activity)

```
INTEGER ABC,I001, X;                 (6.18)
```

by first matching the string INTEGER in the normal manner. The ID syntax macro next matches the ABC after which the commas in the pattern and input are matched. When the closing loop bracket is encountered in the pattern, the pointer into the pattern kept by the processor (hereafter called the pattern pointer) is reset to point to the ID syntax macro call. This time I001 will be matched by the ID macro. When the pattern pointer reaches the exit symbol for the second time the loop is considered to have successfully been matched the minimum number of times (2) specified in its pattern, whereupon the current input scan pointer value is saved and a flag is set. If, after this, further attempts to match the pattern within the loop brackets fail, the input scan pointer will be reset to the saved value and the flag being set will cause the loop construct to signal the completion of a successful match to the surrounding pattern.

As matching continues the commas in the input and pattern are matched and the pattern pointer is reset for the second time. After X matches the ID macro the exit symbol is encountered and the saved input scan pointer is replaced by the current one. The semicolon in the input will now fail to match the comma in the pattern, causing the input scan pointer to back up to its most recently saved value (immediately after the X). The matching process now resumes with the first match item past the loop brackets and the semicolons in the pattern and input are successfully matched. Loop patterns may be nested; thus the pattern

```
'DIMENSION'* <1, ID '(' <1, NUM/','> ')' / ',' >    (6.19)
```

will successfully match any FORTRAN DIMENSION statement. If the exit symbol is omitted in defining a loop construct, the processor will insert one just before the closing loop bracket.

The exit symbol and right loop bracket will delimit the scope of an alternation symbol. The left loop bracket will not delimit the scope of an alternation symbol unless no exit symbol was coded for that loop. Thus the brackets within the loop brackets are redundant in the first line below while in the second they are not.

```
<1, ID / <'+'|'-'>>
<1, <ID | NUM> / ','>
```

This "fault" of the left loop bracket could be removed by having the pattern compiler convert the construct

```
< number, subpattern / subpattern >
```

into

```
        < number, <subpattern> / subpattern >
```

if the first subpattern contained alternation symbols which were not in turn contained by brackets of any type. This may be done if the author determines that it is worth the trouble.

There is an implementation defined limit for the number of nested loops allowed in a single macro pattern which can be easily changed by altering the `$LPNEST` parameter (default value is eight) at processor generation time. The number of macros activated but not yet terminated plus five times the number of loop match items entered but not yet exited is limited to the value of the parameter `$MATCHLOOP`. The default value of this parameter is 200 and is also easily changed.

6.1.8 Backup and Match Retry

Although the input scan pointer may be backed up when alternative portions of a pattern are being matched, the pattern pointer always moves forward through the macro pattern during matching. The one exception to this rule is that of loop matching which has been described above. Thus, except for simple alternation, the matcher will not seek out alternate paths through the pattern if one path fails. Thus the pattern

```
    < ID | ID2:ID. '(' EXPR ')' > '=' EXP2:EXPR ';'    (6.20)
```

will not match the text

```
                  ARRAY(3)=4+A;                  (6.21)
```

because the first `ID` syntax macro call will match `'ARRAY'`, after which the pattern pointer will jump out of the bracketed match item, try to match `'='`, and fail. Since the pattern pointer is now past the alternative in the brackets, that alternative cannot be tried. To avoid this problem, alternative sequences should begin with the match items that will match the most text, one or more of the match items following the alternative sequences can be instead included after each member of the sequence, or optional match items might be used to avoid part or all of the alternation. Each of these three ideas are applied to the above pattern as follows:

```
    < ID '(' EXPR ')' | ID2:ID. > '=' EXP2:EXPR ';'
    < ID '=' | ID2:ID. '(' EXPR ')=' > EXP2:EXPR ';'   (6.22)
        ID ( '(' EXPR ')' ) '=' EXP2:EXPR ';'
```

While in this case the last is preferred for both speed and space, any of these patterns should perform in a satisfactory manner. It should nearly always be possible to write patterns in which backup is unnecessary. (See 13.2, "References" on page 74, item 7)

A description of the syntax rules for macro pattern definition is given in appendix A.

6.1.9 Trigger Macro Patterns

A few restrictions should be observed when writing patterns for trigger macros. A trigger macro pattern must begin with a pattern string which is called the trigger. The trigger may be as short as one character, may not begin with a blank, and may not be nested within brackets, parentheses, or loop brackets. The trigger should have no alternative, that is, any alternation symbol appearing in a trigger macro pattern must be nested within at least one level of brackets, parentheses, or loop brackets. Failure to observe any of these rules will result in a diagnostic and refusal to compile the offending macro.

A trigger macro is called and given control of the scan only when its entire trigger appears in the input. The user should therefore be careful about using trigger macros having very short triggers, because such macros can be more easily called by the input, and once called there is a certain amount of processor overhead involved in passing control of the scan, stacking the environment, etc. even if the macro fails. Another danger arising from short triggers is discussed in 12.0, "Techniques for Macro Definition and Use" on page 69.

6.2 The Pattern Compiler

During macro definition, patterns are thoroughly checked for syntactic correctness and are stored in the macro buffer in a translated form. Syntax macro calls are linked to appropriate locations in the syntax symbol table, whose members will in turn be linked to the syntax macros themselves by the LINK routine when these macros have been compiled. This level of indirection allows macros with references to syntax macros to be compiled before the corresponding syntax macros themselves are defined.

7.0 Macro Replacement Procedure Definition: Introduction, Data Types, and Expressions

Each macro defined consists of an initial pattern and a replacement procedure. As will be seen in the discussion of the SCAN statement in 8.0, "Macro Replacement Procedure Definition: Statements" on page 46, the replacement procedure may contain additional patterns. The replacement procedure is activated upon the completion of a successful match of the pattern with the input text. When a macro is defined a FORTRAN subroutine compiles the replacement procedures from a higher level language, which is hereafter called the replacement base language or just base language, into a reverse polish string. The polish string is then interpreted by another FORTRAN subroutine whenever the replacement procedure is activated. The base language cannot be compiled directly to machine code if the processor is to remain machine independent. In order to keep the compiler subroutine to a manageable size, the only control statement currently implemented in the base language is the conditional branch, or `IF` statement as in

```
IF (I=1) LABEL1;                    (7.1)
```

If the expression is true the statement labelled by `LABEL1` is the next executed, otherwise control passes to the next statement after the `IF`. Since a logical expression evaluates to the integer 1 if true, and unconditional `GOTO` may be coded

```
IF(1)LABEL1;                        (7.2)
```

Since the recognition and passing of a macro definition to the pattern and replacement base language compilers is done under the control of macros, it is natural to allow these macros to process the macro definitions before they are passed on. Although the user is certainly free to write a set of macros to extend the replacement base language anyway that he chooses, a ″standard″ set of base language extension macros is supplied with the processor and appears in appendix B. Although these macros serve several minor purposes, they are primarily designed to implement and check for the correct use of the control structures

```
IF - ELSEIF - ELSE - END IF
FOR - END FOR
WHILE - END WHILE
LOOP - END LOOP                     (7.3)
GO TO
EXIT, NEXT  (for loops)
```

All base language statements except `MEND` are valid statements in the extended language, but some, such as the conditional branch are rarely used. Hereafter references to the IF statement, unless qualified, will refer to the IF construct in the extended language. In the remainder of this section and the two following sections, care will be taken to point out those properties of the replacement procedure language which are implemented by macros in the ″standard″ set and are therefore not connected with the processor in any other way.

The integer, string, and match variables used by the replacement procedure are local to the procedure and temporary; being allocated upon activation and de-allocated when the procedure terminates. The `GLOBAL` and symbol array variables and text produced by `ANSWER` and `RESCAN` statements are not affected by macro activation and termination, although they may be altered by statements within a replacement procedure.

7.1 Integer Data Type

An integer constant is a string of numeric characters containing no blanks, commas, or decimal points. It is interpreted as a decimal integer whose value must lie between implementation defined limits. Integer constants may be preceded by a plus or minus sign, but these do not change the value of the stored constant; they will be interpreted as operators in the normal fashion when the statement containing the constant is executed.

7.1.1 Integer Variables

An identifier appearing in a replacement procedure is assumed to be an integer variable unless it is declared otherwise. An integer variable may assume any value valid for a FORTRAN integer variable on the machine on which STEP is running.

7.2 String Data Type

A string is a sequence of zero or more characters which is treated as a unit. A string constant is written simply by enclosing the characters in the string with apostrophes. If an apostrophe is desired in a string it must be written as two consecutive apostrophes. Thus the string

```
DON'T DO THAT                    (7.4)
```

is written as the string constant

```
'DON''T DO THAT'                 (7.5)
```

String constants may be of any length. A string constant of zero length may be written as two adjacent apostrophes provided that it is not contained in another quoted string.

7.2.1 String Variables

An identifier will represent a string variable if its first appearance in the procedure is in a STRING declaration statement, which is defined later. The length of a string represented by a string variable is not limited and will be equal to the length of whatever string is assigned to it. A string variable can also have the value null, which is not the same as a zero length string, but can be thought of as a string having negative length. All string variables are initialized to null upon procedure entry.

7.2.2 Match Variables

Text matched by the pattern of a macro is accessed in its replacement procedure by means of match string variables, which are hereafter called match variables. A match variable can be used in the replacement procedure like a normal string variable. A match variable is declared in the macro pattern by writing its associated identifier, followed by a colon, in front of a match item, left loop bracket, left parenthesis, or left bracket (except for the left bracket delimiting an alternative sequence of pattern strings following a BAL, ATOMS, or TEXT syntax macro). The matching process will cause the values of each match variable declared in the pattern to be initialized by the replacement text, minus any leading blanks, returned by the match item that it labels. It is not necessary to delete trailing blanks from a match variable because normally none are matched by the corresponding match item.

If a string containing only blanks is matched the appropriate match variable is initialized to a single blank character.

The match variable which labels a syntax macro call will be set to the text returned by that macro. A syntax macro returns all text to the calling macro by means of the ANSWER statement, described below. If a syntax macro call is not labeled it will be assumed to have a label identical to its name. Thus the two patterns

```
          'FOR' ID '=' EXPR 'TO' EXPR 'DO'            (7.6)
     'FOR' ID:ID. '=' EXPR:EXPR 'TO' EXPR:EXPR 'DO'
```

are identical to the processor.

The text returned by a bracketed item, for normal, optional, or loop brackets, is composed of the replacement strings produced by each matching of each match item within the brackets concatenated together in the order in which the matchings occurred. It is possible, for example, to have a match variable represent an entire pattern string including arguments by simply enclosing the string in brackets and labeling the match item thus formed. Although leading blanks are removed from the text returned by the bracketed match item, any blanks between the portions of text returned by the various items contained by the brackets are not.

In each pattern in (7.6) the match variable 'EXPR' is used to label two match items. When match variables are multiply declared the last declaration occurring in the pattern is the one recognized. Thus the text returned by the first call to the EXPR syntax macro will be inaccessible by the replacement procedure for (7.6), although the matching process for the pattern is unaffected. This pattern could instead be written

```
          'FOR' ID '=' EXPR 'TO' EXP2:EXPR 'DO'         (7.7)
```

which allows the text returned by all three macro calls to be accessed.

A single match item may not have multiple labels. Thus

```
                    ID1: ID2: ID                         (7.8)
```

is not legal, while

```
               ID1: <ID2:ID.>                  (7.9)
```

is, since the syntax macro call ID and the bracketed construct are considered to be two different match items by the processor.

Any match variable whose declaration appears inside loop brackets is declared to be an array of strings whose dimension is equal to the number of loops within which it is nested. These match variables must have the appropriate number of subscripts separated by commas in order to be accessed in the replacement procedure. Each subscript must be an integer expression, which is defined below. Following the FORTRAN convention, the first subscript will refer to the innermost loop, and so on. For example, in a replacement procedure for (6.19) the match variable NUM(2,3) will be initialized to the second subscript of the third variable in the DIMENSION statement being matched. Permissible values for a subscript range from one up to the number of times the loop referred to is successfully matched. The highest permissible value for a given subscript in a match array variable is found by substituting an asterisk (*) for that subscript and omitting any subscripts to the left. The array match variable will then have an integer rather than a string value and will equal the subscript limit. Again using (6.19) as an example, NUM(*) and ID(*)

will each be integers whose values will be equal to the number of variables appearing in the DIMENSION statement that was matched, while NUM(*,2) will equal the number of subscripts appearing in the second variable. If the DIMENSION statement did not have a second variable, NUM(*,2) would be zero.

7.2.3 Existence of Match Variables

Any non-array match variable declared in the pattern will exist, but may be initialized to null if its corresponding match item matched no input text. A match variable array element, however, will not exist if any one of the loops in which its declaration is nested does not match successfully the number of times indicated by the appropriate subscripts. Note that while an iteration of the loop containing the declaration must match as a whole, the match item corresponding to a particular variable declared in the loop need not. In other words, it is possible for a match variable array element to exist and be initialized to null.

The number of times a nested loop pattern will match will in general vary with the match iteration of the outer loops, so that the maximum permissible value for a subscript will depend upon the values of the subscripts, if any, to its right. An attempt to access a nonexistent array element will return a null string and a run-time diagnostic. If the leftmost subscript is an asterisk, however, no diagnostic will be issued and the integer value returned will be zero if the remaining subscripts are out of bounds or if the containing loop did not match at all. For example, if only four variables are in a DIMENSION statement matched by (6.19), then NUM(*,5) would be zero, and no diagnostic is issued.

7.3 The GLOBAL Array

GLOBAL is an integer array whose size is implementation defined. A GLOBAL array element may be used anywhere and in the same way that a normal integer variable is used. The values assigned to the GLOBAL array will remain until another assignment is done or a control card (see below) is read. These values are not affected by procedure activation and backups. Each element of the GLOBAL array is initialized to zero when the processor is started.

Values in the GLOBAL array are also changed by control cards of the form

```
/C{INTEGER},{INTEGER}
```

where the beginning slash must be in column one and no embedded blanks are allowed. The second integer may be preceded by a minus sign. The value of the second argument is stored in the GLOBAL array element whose index is the first argument.

The lower limit for GLOBAL array indices is zero while the upper limit is implementation defined. A diagnostic is printed if the index is out of this range in a replacement procedure or control card. The value returned in a replacement procedure would then be the zeroth element of the array.

Elements one to twenty of the GLOBAL array are used as control variables by the processor. These elements can be thought of as switches which have the value one when on and zero when off. The locations currently used are

1. COMPILE. If this switch is turned on by the replacement procedure of a trigger macro then all of the text produced by the ANSWER statements of that procedure is considered to be a macro definition and is passed to the pattern

and replacement procedure compilers. The ANSWER text returned to the caller is then a string of zero length and the switch is turned off. There is no effect upon any RESCAN text produced by the same macro. This switch should not be set by a control card.

2. QUOTE. Normally a comment must be enclosed in double quotes ("). If this switch is on then comments not closed by a double quote before the end of the card will be closed by the processor.
3. TRACE. If this switch is on any ANSWER and RESCAN text produced by a replacement procedure will be dumped into the listing file along with the words "ANSWER" or "RESCAN" and the name of the macro if it is a syntax macro, or its trigger if it is a trigger macro. The normal disposition of the ANSWER and RESCAN produced text is not interfered with.
4. LISTING. The pattern compiler will output the pattern of the macro being compiled to the listing file if this switch is on. The replacement procedure will also be listed on a statement by statement basis. Note that compiler diagnostics are listed regardless of the status of this switch.
5. OBJECT. When this switch is on the object code produced for each statement by the replacement compiler will be listed after that statement.
6. NEST SWITCH. As described in 4.0, "Input Reader" on page 16 it is possible for the input reader to update the nest level count when bracket symbols are encountered. This will be done if GLOBAL(6) is set equal to one.
7. NEST COUNT. The nest count itself is kept in GLOBAL(7) so that it may be easily accessed by the macros. The nest count value is printed in front of each source listing line.
8. INDENTATION AMOUNT. This, as all other elements of the GLOBAL array, is initialized to zero, so that no nest level related indentation appears in the source listing. Each line of the source listing will be indented by an amount equal to the nest level count multiplied by the indentation amount.

Elements 21 to 30 of the GLOBAL array are by convention reserved for the use of macros extending the macro definition language.

7.4 Syntax Descriptions for the Replacement Procedure Language

For the remainder of this manual, STEP macro patterns are used to define the syntax of the various expressions and statements that are introduced. In order to do this, the existence of several syntax macros will for the moment be assumed. INTVAR, STRVAR, and MATVAR will match the name of any integer, string, or match variable, respectively. Any of these macros could be defined using a pattern which consisted of a call to the ID syntax macro and a replacement procedure which looked up the resulting identifier using the symbol table facility, which is described later. If the identifier did not represent the proper class of variable, a FAIL statement, which is also described later, could be executed and the macro would act as if its pattern had never matched the identifier in the first place.

7.4.1 Integer Expressions

An integer expression is a combination of arithmetic operators, integer constants, integer variables, and functions which have integer values. The syntax of and the operators used in these expressions is identical to that allowed by FORTRAN with the exception of the exponentiation operator.

The arithmetic operators, listed in descending priority, are

```
                              + , -    (UNARY OPERATORS)
                              * , /
                              + , -
```

where operators on the same line have equal priorities. An expression may not contain adjacent arithmetic operators, nor may it contain adjacent operands. When two operations in an expression have equal priority the leftmost one is done first. Parentheses are used in the normal manner to change the order of evaluation. Examples of integer expressions are:

```
         1     -A      A+1*4      -(NUM(*,2)+A)*B/(I-8)    (7.10)
```

It is assumed that A, B, and I were not declared as match variables in the pattern and were not declared as string variables in the replacement procedure. Except for declaration statements, an integer expression may appear anywhere that an integer constant would be allowed in a replacement procedure statement.

The pattern INTEXP which matches all integer expressions accepted by the base language compiler is written

```
        MACRO SYNTAX: INTEXP
              <1, TERM / '+' | '-'>;
              ANSWER MATCH;
              END MACRO;

        MACRO SYNTAX: TERM
              <1, FACTOR / '*' | '/'>;                       (7.11)
              ANSWER MATCH;
              END MACRO;

        MACRO SYNTAX: FACTOR
              ('+'|'-') < 'GLOBAL(' INTEXP ')' | INTVAR | NUM |
                  MATVAR '(*' <0, ',' INTEXP> ')' |
                  '(' INTEXP ')' | INTFCN | SYMPNTR | SYMREF>;
              ANSWER MATCH;
              END MACRO;
```

The definition for INTFCN, which matches any function that returns an integer value, will be given in 9.0, "Macro Replacement Procedure Definition: Intrinsic Functions, Listings, and Object Code" on page 57. The definitions of SYMPNTR and SYMREF will be given in the subsection on the symbol array facility. The pattern ('+'|'-') in FACTOR will match both unary operators, while the infix '+' and '-' will be matched in the INTEXP pattern. Again, note that a match variable having an asterisk as its first subscript is treated as an integer. The above macros have been written so as to reflect operator precedence rules.

7.4.2 String Expressions

A string element is defined as a string constant, a string variable, a match variable or array element, or a function which evaluates to a string. String expressions consist of one or more string element separated by the concatenation operator (||), which is the only string operator available. Various string functions that are available, such as substring, will be described in the section on functions. The concatenation infix operator operates only on variables, constants, or expressions having string data type. The result is the creation of a new string formed by joining the operands in the order written. Parentheses are allowed in string expressions and will change the order in which the concatenations are done, but are of little practical use. The priority of the concatenation operator is higher than that of the arithmetic operators although this is seldom of importance since autoconversion between the string and integer data type is never done. Its priority is also higher than those of the relational, logical, and symbol array (described later) operators. If U and V are string variables, the following are valid string expressions:

```
          V    V||U||'ABC'    ('ABC'||V)||U          (7.12)
```

The pattern STREXP to match all string expressions accepted by the base language compiler is written

```
MACRO SYNTAX: STREXP
      <1, <STRELEMENT | '(' STREXP ')'> / '||'>;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: STRELEMENT
      STR | STRVAR | STRFCN | MATREF | SYMREF;    (7.13)
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: MATREF
      MATVAR ( '(' <1, INTEXP / ','> ')' );
      ANSWER MATCH;
      END MACRO;
```

The definition of STRFCN, which matches any function which evaluates to a string, will be given in the section on functions. Again, the definition of SYMREF is given in the subsection on the symbol array facility.

7.4.3 Logical Expressions

A logical expression always evaluates to an integer. The value of this integer may then be interpreted as false (zero) or true (nonzero). A logical expression can be an arithmetic expression, a relational expression, or, in the proper context, a match variable. A logical expression may also be composed of other logical expressions separated by the infix logical operators ″&″ or ″|″, or preceded by the prefix operator ″¬″.

A relational expression is composed of a relational operator with an expression on each side. Both expressions must evaluate to an integer or both must evaluate to a string. The relational operators are

```
        =      ¬=      <      <=      >      >=
```

all of which have the same priority, which is lower than that of the arithmetic and string operators. A relational expression will evaluate to a 1 or a 0, depending upon whether the relation is satisfied or not. Strings are compared by a left to right character by character comparison according to the collating sequence defined by the internal character mapping (see 4.0, “Input Reader” on page 16). In the event that all of the characters in one string are the same as the first characters in another the longer string is considered to be greater. A null string or match variable will always compare as strictly less than a string of zero length. In the future, it may be possible to introduce more flexibility in the ordering of the internal character map in STEP so that the collating sequence at compile time could be adjusted to coincide with that at run time, which will in general be machine dependent.

A match variable is considered to be a logical expression if it appears as the sole argument of an IF statement or is directly operated on by one of the three logical operators. It will evaluate to zero if its value is null, and to a nonzero value if it is not null. This allows for a convenient way to check whether or not a given portion of the macro pattern matched the input or not.

The logical operators are, in order of decreasing priority, "¬", "&", and "|". The priorities of the logical operators are lower than those of the arithmetic, relational, string, and symbol array operators. The result of operation by a logical operator is always 1 or 0, depending upon whether the expression is true or false.

Logical and relational operations of equal priority are performed left to right. Parentheses can be used in the normal manner to change the order of evaluation. Note that the expression

```
                    7 > 6 > 5 < 3                    (7.14)
```

is perfectly acceptable to the base compiler. The operations would be performed from left to right: `7 > 6` evaluates to 1, then `1 > 5` evaluates to 0, and finally `0 < 3` evaluates to 1. The extension macros will not, however, allow such an expression as the argument of an extended IF, ELSEIF, or WHILE statement. A plus or minus sign immediately following a relational or logical operator is interpreted as a prefix operator. If NUM is a two dimensional match variable array, ID is a match variable, S, T, and U string variables and all other variables are of integer type, then example logical expressions can be written as

```
I                    (S < T | I = 6) & J
I=6                  ID > 'ABCD' | NUM(2,3) < ID
NUM(2,3) | I = 6     NUM(*,2) > 6                (7.15)
```

Note that while NUM is used as a logical variable in the third example, it is used as a string variable in the fifth, even though it occurs adjacent to the "|" logical operator. The higher priority of the "<" operator prevents the "|" from directly operating on the NUM match variable. NUM(*,2) is, of course, treated as an integer.

The way in which logical and integer expression have been mixed is perhaps unfortunate but cannot be avoided without introducing a separate logical variable type along with the appropriate declaration statement into the base language compiler. Except for a restriction to binary relational expressions in certain contexts, the extension macros do not place any further restrictions on the mixing of logical/integer expressions.

The pattern LOGEXP which matches all logical expressions accepted by the base compiler is written

```
MACRO SYNTAX: LOGEXP
     <1, LOGTERM / '|'>;
     ANSWER MATCH;
     END MACRO;
```

```
MACRO SYNTAX: LOGTERM
      <1, LOGELEMENT / '&'>;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: LOGELEMENT                          (7.16)
      ('¬') <RELATION | INTEXP | MATREF | '(' LOGEXP ')'>
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: RELATION
      <1, INTEXP / RELOP> | STREXP R2:RELOP S2:STREXP ;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: RELOP
      '=' | '¬=' | '<=' | '<' | '>=' | '>';
      ANSWER MATCH;
      END MACRO;
```

The three calls to INTEXP in the pattern for the FACTOR macro in (7.11) and the call to INTEXP in the pattern of the MATREF macro in (7.13) must now be changed to LOGEXP in order to coincide with the way in which the base language compiler operates. For this reason LOGEXP will appear many places in future syntax definitions where the reader might at first expect to see INTEXP. The text matched by LOGEXP in these cases will still be referred to as an integer expression. Note that the order in which some of the relational operators appear in the pattern for RELOP is important. The restrictions on the use of a match variable as a logical variable are not given by the above macros. In practical work, this extension of the meaning of the match variable has proven very convenient, although it is particularly messy to place into the grammar, or patterns, which describe the language syntax.

8.0 Macro Replacement Procedure Definition: Statements

The statements currently implemented in the base language replacement procedure compiler are the `STRING`, Assignment, `IF`, `OUTPUT`, `FAIL`, `MARK`, `DROP`, `SCAN`, `MERGE`, `RETURN`, and `MEND` statements. All of the above words except Assignment are reserved by the compiler and may not be used in a macro definition for any other purpose. The identifiers `THEN`, `NOTRIG`, `USING`, `NONEW`, and `LOCAL`, which are options of various statements, are also reserved. The statements implemented by the extension macro set are discussed after those of the base language.

8.1 STRING Statement

The `STRING` statement can be matched by the pattern

```
'STRING' <1, ID> ';' ;
```

All of the identifiers appearing in a `STRING` statement are declared to be variables having string data type, and must not appear in any statement previous to the `STRING` statement. The compiler will issue a diagnostic and the macro will be deleted after compilation if any variable appearing in a string statement has appeared previously anywhere else in the macro definition. The `STRING` statement produces no executable code.

8.2 Arithmetic Assignment Statement

A pattern to match an arithmetic assignment statement can be written

```
<'GLOBAL(' LOGEXP ')' | INTVAR> '=' LOGEXP ';' ;
```

The expression on the right side is evaluated and stored into the integer variable. Automatic type conversion between integer and string data types is not done upon assignment or under any other conditions. The following are valid arithmetic assignment statements:

```
I=6    I=6 *(J+K/3)    I=I=6 | NUM(*,2)       (8.1)
```

8.3 String Assignment Statement

A string assignment statement will match the pattern

```
<STRVAR | MATREF> '=' STREXP ';' ;
```

The quantity on the right hand side is evaluated to form a string which is thereafter represented by the variable on the left. Note that the match variables are only initialized by the pattern matching process, and may be reset to other string values in the replacement procedure. Any non-array match variable declared in the pattern exists and can have values assigned to it, even if the match item it corresponds to did not match anything. Match variable array elements can have strings assigned to them only if they exist.

8.4 IF Statement - (Base Language)

The format of the base language IF statement is

```
                    'IF(' LOGEXP ')' ID ';' ;
```

The arithmetic or logical expression is evaluated. If it is true (nonzero) control is transferred to the statement which is preceded by the label. If it is false control transfers to the next statement after the IF. If a match variable appears alone as the IF expression it will be evaluated as a logical expression as described above. Thus if NUM and ID are match variables

```
                       IF (NUM) LABEL1 ;                (8.2)
```

is legal whereas

```
          IF (NUM+ID) LABEL1;      IF(NUM*6) LABEL;      (8.3)
```

are not. Example IF statements are:

```
          IF(3 = I) LAB1;       IF(38-I*(8+J)) LAB3;
          IF (NUM(3,2) > STRING||STR2) LAB2;             (8.4)
          IF( ¬ NUM(3,2)) LAB1;
```

where the last of these makes use of the match variable as a logical variable. While the IF statement may be used in the extended language, it almost never is. If it must be, the target label should never be the identifier THEN!

The base language compiler does not recognize a GO TO statement. Instead of the statement

```
                       GO TO LABEL1;                    (8.5)
```

one may code

```
                       IF(1) LABEL1;                    (8.6)
```

8.5 SCAN Statement

The SCAN statement can appear in either of the two following forms:

```
          'SCAN' MACRO_PATTERN ';' ;
          'USING' STREXP 'SCAN' MACRO_PATTERN ';';       (8.7)
```

The two types of SCAN statement shown above will be called types one and two, respectively. The SCAN statement allows the replacement procedure to gain direct access to the STEP pattern matcher. The type one SCAN statement causes the input text, starting from the current location of the input scan pointer, to be matched against the macro pattern following the keyword SCAN. The match variables declared in the pattern are initialized by the results of the match just as they are for the first pattern in the macro. In the following macro definition the syntax macros NORMAL_LIST and CONVRT_LIST will each match a list of items following a DIMENSION statement, but will generate different code. The value of a flag stored in GLOBAL(25) indicates which of these syntax macros is to be used.

```
MACRO SYNTAX: DIMENSION
      'DIMENSION';
      ANSWER MATCH;
      IF GLOBAL(25)=1 THEN
          SCAN NORMAL_LIST;                    (8.8)
          ANSWER NORMAL_LIST;
      ELSE
          SCAN CONVRT_LIST;
          ANSWER CONVRT_LIST;
      END IF
      END MACRO;
```

In the next example, the syntax macro STATEMENT is assumed to exist and to match any statement of a given language, after which it writes the statement, perhaps in a modified form, directly to the code file via the OUTPUT statement. A simple macro to match a subroutine can then be written

```
MACRO SYNTAX: SUBROUTINE
    'SUBROUTINE' ('('ARGLIST')')';';
    OUTPUT MATCH;
    SCAN <0, STATEMENT>;
    SCAN END: 'END;' ;                         (8.9)
    IF ¬END THEN
        WARN 'MISSING END INSERTED';
    END IF
    OUTPUT 'END;';
    END MACRO;
```

After the subroutine declaration and argument list have been matched they must be immediately inserted into the code file so that they will appear before the code generated by the calls to the STATEMENT macro, so the SUBROUTINE macro must ″break out″ of its pattern matching process in order to do this. The scan statement in this case simply allows the matching process to resume where it left off. Note that a subroutine containing thousands of statements can be matched as a unit by the above macro, since the text produced by the matching process is not stored in the processor's buffers, as it would be if STATEMENT ANSWERed all of its text back to SUBROUTINE.

The concept of a distributed pattern makes it necessary to modify the definition of a STEP macro as a pattern followed by a replacement procedure. A macro definition then consists of pattern matching and other replacement procedure statements. The first statement of a macro must be a pattern and is called the initial pattern. The initial pattern must always be successfully matched against the input if the remaining statements of the macro are to be executed. If a subsequent pattern is activated by a type one SCAN statement and fails to match, the input scan pointer is reset to its value before the abortive match started and all of the match variables declared in that pattern are set to null. Except for this action the execution of the replacement procedure continues with the next statement. The syntax of the pattern appearing in each SCAN statement must follow the rules already specified for the initial pattern of a syntax macro. Match variables must not appear in a replacement procedure before they are declared in either the initial pattern or a SCAN statement. If the same match variable name is declared more than once in any pattern, or in more than one pattern, a warning diagnostic is printed and the match variable will refer to the text matched by the last match item appearing in the replacement procedure which corresponds to that match variable. Any match variable that has not matched text or is nested within part of a pattern that failed to match text will be initialized to null. A pattern in a SCAN statement may be repeatedly matched against the input. Thus the previous example can be rewritten

```
MACRO SYNTAX: SUBROUTINE
    'SUBROUTINE' ('(' ARGLIST ')') ';';
    OUTPUT MATCH;
    LOOP
        SCAN STATEMENT;                          (8.10)
        IF ¬STATEMENT THEN EXIT; END IF
    END LOOP
    SCAN END:'END;';
    etc.
```

All match variables in the SCAN statement's pattern will reflect the progress of each successive match as if it were the first. Note that if the match variable STATEMENT exists, it will be a string of length zero since STATEMENT does not answer any text, but only does OUTPUTs. If for some reason STATEMENT does ANSWER text back to SUBROUTINE, that text is deleted just before the same SCAN statement is executed again. This is necessary if the STATEMENT match variable is to be initialized to the text to be matched in the next iteration, or initialized to null if the text cannot be matched. If STATEMENT produces RESCAN text then that text is placed into the input exactly as it would be by (8.9). The production of RESCAN text in this particular instance would cause a call to the STATEMENT macro in either type of loop to scan the text produced by its predecessor.

The type two SCAN statement causes the scan pointer to be temporarily redirected to the string resulting from the evaluation of the string expression following the keyword USING. This string is matched exactly as if it were the entire input to the processor; trigger macros can be activated and syntax macro calls nested to any level during the scan. The end of the string is treated as the end of input, so that any pattern attempting to move the input scan pointer beyond the end of the string will be forced to fail. The type two SCAN statement does not affect the values of the MATCH and SOURCE functions (these are defined in the next section). When the pattern of the SCAN statement, successfully or unsuccessfully, completes matching, the scan pointer reverts to the input and is reset to the value it held before the type two SCAN statement was encountered.

8.6 ANSWER Extension Statement

The format of the ANSWER statement is

```
'ANSWER' ('NOTRIG') <1, STRELEMENT> ';' ;
```

where the keyword NOTRIG is optional and each of the string elements when evaluated have string data type. As seen above, a string element is a string expression with the exception that no operators that are not nested within at least one level of parentheses are allowed. The only operator which could be used in violation of this rule is the string concatenation operator, whose use in this way in an ANSWER statement would be redundant. If after its evaluation, a string element is neither null nor of zero length and does not have a leading blank, one is inserted. The elements are concatenated together in the order of their appearance to form one string. Before the replacement procedure terminates, all text strings thus formed are concatenated in the order in which their corresponding ANSWER statements were executed. The resulting text is returned to the calling macro if the macro that is terminating is a syntax macro. If it is a trigger macro the text is simply deleted unless it is to be passed to the macro definition compilers.

The insertion of the leading blanks in front of any item is suppressed by placing a dot (.) in front of that item. If S is a string variable and ID and NUM are match variables and ID='FORTRAN', NUM='123', and S='T' then

```
      ANSWER 'L'.NUM ID;       produces     ' L123 FORTRAN'
      ANSWER .S'NOW' S S.S;    produces     'T NOW T TT'   (8.13)
```

It is necessary to separate two adjacent match or string variables and two adjacent string constants appearing in an ANSWER statement by blanks in order to avoid confusion as to their meaning. It is not necessary to insert blanks between a string constant and string variable or any other two items which can be distinguished. The NOTRIG option is described with the definition of the RESCAN statement below.

The ANSWER statement is actually an extended statement, and will not be recognized by the base language compiler. The only conversion made by the macro definition macros is to convert 'ANSWER' to 'OUTPUT(1)', however. See the section on the OUTPUT statement for a further description.

8.7 RESCAN Extension Statement

The format of the RESCAN statement is the same as that for the ANSWER statement except that the keyword 'ANSWER' is replaced by 'RESCAN'. One text string is produced from all of the RESCAN statements executed in the replacement procedure in the same way as for the ANSWER statement, but this text string then effectively replaces the text which the macro pattern matched, after which the input scan pointer is set to its beginning. Subsequent text matching will commence with this ″replacement″ text being scanned. Should the input scan pointer ever back up past the beginning of this replacement text, however, the text matched by the pattern which produced the replacement text, which may itself be original input or replacement text generated earlier, will be reinstated. Both trigger and syntax macros may produce RESCAN text, although the majority of it will normally be produced by the trigger macros. ANSWER and RESCAN statements may appear in the same replacement procedure and will not interfere with each other.

If the NOTRIG option is coded for either an ANSWER or a RESCAN statement the text produced is altered so that it can never cause trigger macro invocation. In all other respects, however, this alteration of the text is undetectable. The altered text can then be matched in the normal way by any match item of any macro pattern except for the trigger of a trigger macro pattern. A trigger can never match altered text. The alteration is permanent, so that even if the altered text is broken up or assigned to string variables in a replacement procedure and passed up through several levels of macro calls via ANSWER statements before being placed into the input via a RESCAN statement, the text originally altered will remain so. For example, it may be desirable to abbreviate calls to a certain subroutine which nearly always has the same arguments with the help of the following macro.

```
MACRO TRIGGER: 'CONVRT' ;
       RESCAN 'CALL' ;
       RESCAN NOTRIG 'CONVRT';          (8.14)
       RESCAN '(132,I,J)';
       END MACRO;
```

If text produced using the NOTRIG option is underlined, the statement CONVRT becomes CALL <u>CONVRT</u>(132,I,J) with the input pointer reset to the beginning of CALL. When the scan resumes and the scan pointer finally reaches CONVRT, the reinvocation of the above trigger macro will now be inhibited. The macro

```
                MACRO TRIGGER: 'CALL' 'CONVRT' ;          (8.15)
                         etc.
```

would match the text produced by (8.14), however, because a trigger macro is called by the appearance of its trigger in the input, after which matching proceeds without regard to altered text. Altered text need not start or stop on atom boundaries, nor can it define atom boundaries. Thus the syntax macro

```
                MACRO SYNTAX:BB
                         ID ;
                         ANSWER NOTRIG 'DECL';            (8.16)
                         ANSWER .'ARE';
                         END MACRO;
                         etc.
```

will match any identifier and return the single atom DECLARE, where the altered text is again underlined, to the calling macro. If a macro to call BB is defined

```
              MACRO TRIGGER:
                       'START' BB ;
                       STRING S,T;
                       S=SUBS(BB,1,5); T=SUBS(S,2);       (8.17)
                       RESCAN T.'ABC';
                       END MACRO;
```

The text returned to the input when this trigger macro successfully matches the input would be

```
                                 ECLAABC                 (8.18)
```

where the altered text is again underlined.

When text altered by the NOTRIG option is printed by the trace facility it will be mapped through the second output character set as described in 4.0, “Input Reader” on page 16.

As with ANSWER, the RESCAN statement is actually an extended statement and will not be recognized by the base language compiler. The only conversion made by the macro definition macros is to change 'RESCAN' to 'OUTPUT(2)', however.

8.8 OUTPUT Statement

The format of the OUTPUT statement is

```
   'OUTPUT' ( '(' LOGEXP ')' ) ('NOTRIG') <1, STRELEMENT> ';';
```

If LOGEXP is present and evaluates to an integer between four and ten then on a statement by statement basis the string elements are concatenated exactly as for the ANSWER statement, with the dot (.) having the same significance, and the resulting text is written to an output file. Text is written immediately after each OUTPUT statement completes execution, not when the replacement procedure terminates. If the integer expression is not present or evaluates to four then the text is written to the code file. If the integer expression is present and evaluates to a number between six and ten the text is written to one of the five auxiliary files. The auxiliary file numbers six through ten correspond in order of appearance to the five auxiliary file unit numbers on the card following the output map initialization card.

If LOGEXP is present and evaluates to one, two, or five, then the OUTPUT statement behaves as an ANSWER, RESCAN, or WARN statement, respectively. In fact, the ANSWER, RESCAN, and WARN statements are just shorthand for particular forms of the OUTPUT statement. Thus it is possible to have one OUTPUT statement behave as an ANSWER, RESCAN, WARN, or OUTPUT statement simply by changing the internal "unit" number. Throughout this manual, if it is not qualified, the word OUTPUT will refer to the default form of the OUTPUT statement which writes directly into the code file.

If an initial pattern or the pattern of a type one SCAN statement fails to match the input at some point, the input scan pointer is backed up to the beginning of the text that was to be processed by the offending pattern. Any text produced by the ANSWER or RESCAN statements of the macros implicitly or explicitly called during the abortive match disappears. Text produced by types four to ten of the OUTPUT statement will not, however, disappear because it has already been written to an external file. Since it is not logically consistent for a macro to cause OUTPUT text to be generated and then fail, back up, and allow some other macro to independently match the same input and produce output of its own, in the future the following restriction may be placed into the processor. The input scan pointer would not be allowed to back up past or into the input (or RESCAN) text that was matched by any initial pattern or type one SCAN statement before the last type 4-10 OUTPUT statement has been executed in any macro replacement procedure. Any pattern match failure or FAIL statement (described later) execution attempting an illegal backup would be trapped by the processor and a diagnostic would be issued. The position of the scan pointer would not be changed. Except for this action, processing would continue unaffected. Meanwhile, it is therefore important that some macros be written so that when their execution passes a certain point, they never fail. Consider, for example, the following macro

```
MACRO SYNTAX: MAIN_PROGRAM
    <0, DECLARE_STATE>
    <0, EXEC_STATE>
    'END;' ;                                  (8.19)
    OUTPUT 'END;';
    END MACRO;
```

This macro will match a main program consisting of declaration statements, followed by executable statements, which in turn must be followed by an END statement. The DECLARE_STATE and EXEC_STATE macros both OUTPUT the text that they match in a possibly altered form. The EXEC_STATE macro would look something like

```
MACRO SYNTAX: EXEC_STATE
    ¬<'END;'>
    <IF_STATE     |
    ASSIGN_STATE  |
    OTHER_STATE   |                         (8.20)
       .
       .
       .
    TEXT ';'      >;
    (replacement procedure)
    END MACRO;
```

EXEC_STATE matches anything but an END statement. If the user neglected to insert an END statement at the end of his code, then the EXEC_STATE loop in MAIN_PROGRAM will match right up to the end of input, where even TEXT will fail, then the 'END;' pattern string in MAIN_PROGRAM will attempt to match and fail, causing MAIN_PROGRAM to fail. The processor will then generate a diagnostic when the scan pointer tries to back up to the beginning of the first declaration statement that was matched. The solution to this situation is to replace the pattern string in MAIN_PROGRAM by the pattern END:('END;') and generate a WARN diagnostic if the match variable END does not exist. The OUTPUT statement will then insert the END; into the code file anyway and MAIN_PROGRAM would match the entire input successfully. Most of the processor generated diagnostics, such as the one described above, exist in order to help the macro writer debug his macros and should never be seen by the end users of those macros.

8.9 WARN Extension Statement

The format and function of the WARN statement on a statement by statement basis is again identical to those of the ANSWER statement with the exception of the keyword change. As each WARN statement ends execution, the text formed by it is output to the listing file, thus allowing the macro writer the ability to compose and list out diagnostic messages or any other useful information that he desires.

As with ANSWER, the WARN statement is actually an extended statement, and will not be recognized by the base language compiler. WARN is a particular instance of the OUTPUT statement whose argument is 5.

8.10 MERGE Statement

The MERGE statement has the format

```
'MERGE' <1, LOGEXP / ','> ';';
```

Starting from the left, each integer expression is evaluated to form a number that must be between six and ten. The corresponding auxiliary file is then rewound and copied into the code file. After copying is completed each auxiliary file is again rewound and is ready to be reused. The MERGE statement will probably soon be replaced by more general file handling statements which will in addition provide for multipass processing and one or more forms of INCLUDE statements.

8.11 FAIL Statement

The FAIL statement has the format given by the pattern

```
'FAIL;' ;
```

The replacement procedure for a macro will not be activated unless its corresponding initial pattern has matched the input successfully. If, before any type one SCAN statement is executed, a replacement procedure executes a FAIL statement, it is immediately terminated with no ANSWER or RESCAN text being produced, the input scan pointer is reset to the beginning of the text scanned by the initial pattern, and the macro containing the FAIL statement informs the surrounding pattern that it has failed to match. Thus the macro having the pattern

```
'BEGIN' ID 'END' ;                (8.21)
```

could behave exactly as (6.15) if the following statements appeared at the beginning of its replacement procedure:

```
IF  ID = 'FORTRAN' | ID = 'ALGOL'  THEN
    FAIL;                                     (8.22)
END IF
```

If a type one SCAN statement was executed before the FAIL, then the input scan pointer is reset to the beginning of the input text that the SCAN statement matched. If the pattern of that SCAN statement had failed to match in the first place, the input scan pointer is not altered. The execution of the replacement procedure then continues with the next statement. FAIL allows the macro to "undo" the previous pattern match if the replacement procedure determines that such action is necessary. The execution of a type two SCAN statement has no effect on the subsequent execution of a FAIL statement.

8.12 RETURN Statement

The format of the RETURN statement is

```
'RETURN;';
```

Execution of the replacement procedure is terminated and the text strings produced by the ANSWER and RESCAN statements are concatenated and stored. If the trace switch is set the text produced by the ANSWER and RESCAN statements is also output to the listing file.

8.13 MEND Statement - (Base Language Only)

The format of the MEND statement is

```
'MEND;' ;
```

This statement performs the same functions as the RETURN statement, and in addition serves as a compiler directive indicating the end of the replacement procedure. The MEND statement should never appear in a macro written in the extended language as it is generated automatically by the END MACRO statement which is described later.

8.14 Null Statement

A Null statement is written by simply coding a semicolon. Its function is similar to that of the FORTRAN CONTINUE statement. It may be labeled like any other statement and multiple Null statements may be written. Thus

```
LAB1: LABEL2: ;;; LABEL3:;A=B ;          (8.23)
```

is a legal construction and all three labels will refer to the beginning of the statement A=B;.

8.15 IF Extension Statement

A pattern matching the IF extension statement can be written

```
'IF' LOGEXP 'THEN' <0, STATEMENT>
<0, 'ELSEIF' LOGEXP 'THEN'
                    <0, STATEMENT>
('ELSE' <0, STATEMENT>)
'END IF' ;
```

The syntax macro STATEMENT is assumed to have been defined to match any statement of the extended macro definition language. Again, the extended language includes all base language statements except MEND. The rules for evaluation of the logical expression in the IF and ELSEIF clauses are are slightly different from those for the base language IF statement. These rules are given by the LEXPR macro in appendix B, although the NLEXP macro, also in appendix B, is actually used for scanning the logical expressions in IF statements. The two identifiers END IF are always required as a termination for the extended IF statement. The IF - ELSEIF - ELSE - END IF construct with all contained statements (some of which might be other extended IF statements) is considered to be a single statement of the extended language and will be matched by the STATEMENT macro. Note that ELSEIF is a single identifier; when split into two it becomes an ELSE clause whose first statement is an IF. This is quite legal, although an extra END IF is required to close the new IF statement.

8.16 FOR Extension Statement

A pattern matching the FOR statement can be written

```
'FOR' INTVAR '=' LOGEXP ('TO' LOGEXP) ('BY' LOGEXP) 'DO';
   <0, STATEMENT>
'END FOR' ;
```

INTVAR matches the loop index, which will be initialized to the value of the first integer expression and thereafter incremented by the value of the third integer expression. Before each iteration the value of the loop index is compared with that of the second integer expression. Control is transferred to the first statement following the FOR loop if the value of the loop index is greater. If the third integer expression is absent then the loop index is incremented by one. If the second is absent looping continues until terminated by a GO TO, RETURN, or EXIT statement.

8.17 WHILE Extension Statement

A pattern matching the WHILE statement is written

```
'WHILE' LOGEXP 'DO'
    <0, STATEMENT>
'END WHILE' ;
```

Looping will continue as long as the logical expression, which is checked before each iteration, remains true. The rules for evaluating the logical expression in the WHILE statement are the same as those for the IF extension statement.

8.18 LOOP Extension Statement

A pattern matching the LOOP statement is written

```
'LOOP' <0, STATEMENT> 'END LOOP' ;
```

Looping will continue until terminated by execution of a GO TO, RETURN, or EXIT statement.

8.19 GO TO Extension Statement

The GO TO statement is matched by

```
'GO TO' ID ';' ;
```

A base language statement of the form `IF(1) ID;` is generated.

8.20 EXIT Extension Statement

The EXIT statement is matched by

```
'EXIT;';
```

and must appear within the range of a FOR, WHILE, or LOOP statement. EXIT will cause a transfer of control to the first statement following the innermost loop within which it is nested.

8.21 NEXT Extension Statement

The NEXT statement is matched by

```
'NEXT;';
```

and must appear within the range of a FOR, WHILE, or LOOP statement. NEXT causes the next iteration of the innermost loop within which the NEXT statement is nested to be commenced immediately.

9.0 Macro Replacement Procedure Definition: Intrinsic Functions, Listings, and Object Code

The functions currently implemented in the replacement procedure compiler are SUBS, INDEX, LEN, CS, CN, MATCH, SOURCE, and SCANOK. All of the above identifiers are reserved by the compiler and must not be used in a macro definition for any other purpose. The word GLOBAL, which can be thought of as representing a function, is also reserved. Function arguments may be any expressions, including those involving other functions, that are allowable by the compiler and whose data type is correct. Arguments are first evaluated, after which the function is evaluated.

```
'SUBS(' STREXP ',' LOGEXP (',' LOGEXP) ')'
                  result is string
```

SUBS produces a substring of the first argument. The second argument specifies the starting column of the substring while the third argument specifies its length. If the third argument is omitted the remainder of the string after the starting position is returned. For example, SUBS('FORTRAN',4,3) returns the value 'TRA', while SUBS('FORTRAN',4) returns 'TRAN'.

If the second argument evaluates to less than one, it is set equal to one. If the third argument is less than zero it is set equal to zero. Any portions of the resulting substring which extend beyond the boundaries of the first argument are truncated. If the entire substring is specified to be outside of the first argument a zero length string is returned.

```
'INDEX(' STREXP ',' STREXP ')'
         result is integer
```

INDEX returns the starting position of the second argument within the first. If the second argument is not a substring of the first the value zero is returned. For example, INDEX('FORTRAN','RTR') returns the value 3.

```
'LENGTH(' STREXP ')'
  result is integer
```

LENGTH returns the length of the string resulting from the evaluation of its argument.

```
'CS(' LOGEXP ')'
result is string
```

CS returns a string of digits corresponding to its argument. If the argument is negative a minus sign is appended on the front. There are no leading, trailing, or embedded blanks in the strings returned by CS.

```
'CN(' STREXP ')'
result is integer
```

The argument must evaluate to a string of digits, with an optional plus or minus sign. Leading, trailing, and embedded blanks are allowed, but the string must contain at least one digit. If the argument conforms to the conditions it is converted to an integer value. If the conditions are not satisfied the integer result is set to zero and a run-time diagnostic is printed along with the value of the argument.

```
    'MATCH'
result is string
```

MATCH is actually a match variable, and returns a result identical to that which would be returned if the entire initial pattern were placed in brackets and labelled. Its value is thus all text strings returned by the syntax macro calls and pattern strings in the initial pattern of the macro concatenated in the order in which these items were matched. MATCH can be used in any way that an ordinary match variable can be used.

```
                          'SOURCE'
                      result is string
```

The original text matched by the initial pattern of the current macro or, if any have been executed, the latest type one SCAN statement, is returned. This text may be the original input to the processor or may be composed in part or entirely of rescan text produced by trigger macro calls occurring any time before the pattern in question was activated, but will not contain any rescan text produced after this time. Assignments can be made into MATCH, but not SOURCE.

```
                          'SCANOK'
                      result is integer
```

The function SCANOK, which has no arguments and can appear anywhere (except in declaration statements) that an integer constant is allowed, will return the value true or false as a result of the execution of a type two SCAN statement. SCANOK will be true only when the SCAN statement's pattern matches the entire string. It is then possible for the pattern to match part of the string successfully and SCANOK to be false. For example, SCANOK is true for the first statement that follows and false for the second.

```
                     USING '123' SCAN NUM ;
                    USING '123ABC' SCAN NUM ;            (8.11)
```

In the second statement, the call to the NUM syntax macro successfully matches '123', but does not match the entire string. Note that because the NUM macro will not match a trailing blank, SCANOK is false after the first of the two following statements complete execution and true after the second.

```
                     USING '123 ' SCAN NUM ;             (8.12)
                     USING ' 123' SCAN NUM ;
```

The value of SCANOK for a particular replacement procedure is not defined if the last pattern matched was the initial pattern or that of a type one SCAN statement.

9.1 Symbol Array Facility

A symbol array is basically a one or two dimensional array in which the first subscript may be an arbitrary string having length greater than zero. Macros recognizing a reference to any symbol array variable can be written

```
MACRO SYNTAX: SYMREF
      ID '(' < '@' STREXP | SYMPNTR | LOGEXP>
                 ( ',' LOGEXP ) ')' ;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: SYMPNTR
      ID <('.NONEW')('.LOCAL')|('.LOCAL')('.NONEW')>
              '@' STREXP;
      ANSWER MATCH;
      END MACRO;
```

Thus, to look up the fourth attribute of the identifier VAR and store it in the integer I, one would code

```
                    I=THISYM(@'VAR',4);
```

where THISYM is the name of the symbol array being used and any string expression may follow the @ operator. If, on the other hand, the replacement procedure decides that the fourth attribute of the identifier matched by ID must be equal to K, the attribute can be set by

```
                    THISYM(@ID,4)=K;
```

If no entry for the string represented by ID exists in THISYM, one will be created by the above statement.

A symbol array may be declared at any point in a replacement procedure as long as its declaration occurs before any usage. All symbol arrays are global in scope. If the replacement procedures of two separate macros declare symbol arrays having the same name, then they will both access the same symbol array. The first time a particular symbol array is declared in a replacement procedure space is allocated and initialized to hold it and its name is entered into the symbol array directory (which is actually just another symbol array). Subsequent declarations of the same symbol array in other macros will be required to be identical to the first one.

A pattern to match a symbol array declaration can be written

```
      'SYMBOL' <1, ('STRING') ID '(*' (','NUM) ')'/','> ';'
```

The following declaration statement gives an example of each of the four possible types of symbol arrays

```
            SYMBOL TABLE(*), MAT(*,6),
                STRING RAT(*), STRING CAT(*,7);
```

Each element of TABLE is a single integer which is indexed by a string expression (indicated by the asterisk). Each element of RAT is a single string of arbitrary length. The symbol array MAT is perhaps closer to the type of symbol table a compiler might use. Each element of MAT consists of six integers, which can be indexed by the second subscript. This subscript must be an integer expression whose value can range from one to six. Each of the elements of the CAT symbol array will consist of seven strings. A symbol array variable may be used anywhere that an integer or string variable of the same type is allowed.

9.2 Symbol Array Pointers

A symbol array pointer is an integer. The string preceded by the "@" operator which appears as the first subscript of a symbol array is evaluated to an integer before being used to index the array. If more than one numerically subscripted item in a single symbol array element are to be accessed, time and macro storage space can usually be saved by making use of a pointer variable. Thus, instead of the sequence

```
                    A=MAT(@'ABC',1);
                    B=MAT(@'ABC',2);
                    C=MAT(@'ABC',3);
                STRING RAT(*), STRING CAT(*,7);
```

one should code

```
K=MAT@'ABC';
A=MAT(K,1);
B=MAT(K,2);
C=MAT(K,3);
```

Thus the symbol array name directly followed by the "@", or symbol array, operator, which in turn is directly followed by a string expression, will evaluate to an integer which can be used as the first index of the symbol array. In case the reader is interested, the expressions

```
MAT(@'ABC',1)   and   MAT(MAT@'ABC',1)
```

are identical as far as the processor is concerned.

It is lookup and not assignment that forces allocation of a new element of a symbol array. All integers stored in a newly allocated symbol array element will have the value zero and all strings will be null strings. At times it is necessary to see whether a given element is contained in a symbol array without creating it if it is not there. This can be done by adding the suffix 'NONEW' which is separated by a dot from the symbol array name. The expression

```
TABLE.NONEW @ 'ABC'
```

will be nonzero and point to the element 'ABC' in TABLE if that element exists. Otherwise it will evaluate to zero. The expression

```
TABLE @ 'ABC'
```

will never evaluate to zero and will always point to the element 'ABC' in TABLE, even if that element did not previously exist.

The NONEW suffix is allowed only in expressions which determine a symbol array pointer, and is not allowed in an actual reference to a symbol array element. Thus the statement

```
I=TABLE.NONEW('ABC');
```

will be flagged as an error by the compiler.

The priority of the symbol array operator '@' is lower than that of the concatenation operator, so that in the expression

```
TABLE@U||'ABC'||V
```

the two concatenations will be done before the resulting string expression is converted into an integer index into the TABLE array.

Great care should be taken in using symbol array pointers as their run-time values are not checked. The use of pointers can, however, offer great flexibility in symbol table construction since elements of one symbol array may contain pointers into other symbol arrays.

9.3 The MARK and DROP Statements

Patterns to match a MARK and a DROP statement can be written

```
'MARK' ID ';' ;
'DROP' ID ';' ;
```

These statements allow segmentation of symbol arrays. Each time a MARK statement for a particular symbol array is executed, the tree data structure for that symbol array (the current tree) is placed on a stack and a new tree is allocated and becomes the current tree. When a string is to be looked up, the current tree is searched first, then the tree on the top of the stack, then the next tree on the stack, and so on until a match for the string is found. If no match is found and NONEW was not specified then a new element is allocated in the current tree. The suffix LOCAL enables the search for a matching string to be confined to the current tree. The suffix NONEW may be used along with LOCAL if desired (LOCAL and NONEW may appear in either order). Thus

```
I=TABLE.LOCAL.NONEW('ABC');
```

will cause only the current tree of the symbol array TABLE to be searched for the string 'ABC' and will evaluate to zero of it cannot be found there.

Each time a DROP statement is executed for a given symbol array, the current tree data structure for that array is deleted and the space that it occupied is reclaimed. A tree is removed from the top of that symbol array's stack and becomes the current tree. It is possible to DROP a symbol array even if it has never been MARKed, because when a symbol array is first declared it is MARKed in the process in order to allocate space for its first tree. In this case all of the elements of that symbol array are deleted and the array cannot again be used until it is MARKed.

The keeping of symbol tables for the compilation of code for block structured languages is one example of the usefulness of the MARK/DROP feature of the symbol arrays.

9.4 Replacement Procedure Compiler Listings

When the listing switch (GLOBAL(4)) is equal to one a listing is produced of the replacement procedure being compiled. One statement is listed on each line and is preceded by the line number, which starts at one for the first statement compiled, and the relative offset. The offset gives the number of storage locations from the first storage location to be used by the macro being compiled to the first of the locations in which the statement is stored. Run-time diagnostic messages will give the relative offset at which an error has occurred during procedure execution so that a compile listing can be used, perhaps with the aid of an object listing (produced when GLOBAL(5)=1), to pinpoint the source of the error. Any error found in a replacement procedure statement is listed immediately after that statement.

9.5 Object Code

The replacement compiler produces reverse polish code which is interpreted by another subroutine in the STEP processor. During replacement procedure execution each operand will be stacked and each operation executed as it is encountered. All operands and operators are numbers between 0 and 63 so that it is possible to store them as characters on most machines. If necessary, those few numbers which are outside the above range (such as some integer constants) are split up in an implementation defined way for storage into characters.

After each statement is compiled it undergoes a trial execution to insure that none of the run-time stacks will overflow, that no data type inconsistencies exist, and that the statement can be properly evaluated. A statement error detected before this trial evaluation is flagged as a syntax error, and no object listing can be produced. An error found by the evaluation, which occurs after the object listing is produced, will be flagged as a statement structure error. Some errors, such as subscripting errors, cannot be caught at compile time. Errors detected at compile time will usually force the compiled macro to be deleted from the macro storage buffers, although the compilation will continue on to completion in order that any other errors in the macro might be detected.

9.6 Efficiency

The following rules are recommended in order that replacement procedures execute faster and occupy less space. A match variable used more that two or three times should be assigned to a string variable which then can be used in its place. The same applies to a match array element used more than once. Needless assignments should not be done: if a complicated integer expression is to be evaluated only once and used as an array subscript or function argument it may itself appear in the index or argument list.

9.7 Syntax Summary

A complete description of the syntax requirements for writing replacement procedures is given in appendix A.

10.0 Macro Definition

If GLOBAL(1) is turned on in the replacement procedure of a trigger macro, all of the text produced by the ANSWER statements of that procedure is considered by the processor to be a macro definition. Such a definition must consist first of the keyword 'TRIGGER' followed by a colon for a trigger macro definition, or of the keyword 'SYNTAX' followed by a colon and the name of the syntax macro. Following this, an optional protection level (see below) may be coded within parentheses. Next must come the macro pattern which is terminated by a semicolon. The final item in the definition is the base language replacement procedure, which must contain an MEND statement as its last statement. The syntax for a complete base language macro definition is given in appendix A.

10.1 Local Trigger Macros - Scope Rules

A trigger macro will be local to another syntax or trigger macro if its definition is nested within the other macro's definition. Thus the first of the three macros in (2.21) could be defined with two trigger macros local to it by writing

```
MACRO SYNTAX: EXPRESSION
       TERM '+' EXPRESSION | TERM2:TERM ;
       MACRO TRIGGER: 'PARM1';
               RESCAN '123';
               END MACRO;
       MACRO TRIGGER: 'PARM2';
               RESCAN '321';                  (10.1)
               END MACRO;
       IF TERM2 THEN
           ANSWER TERM2 ;
       ELSE
           ANSWER TERM ',' EXPRESSION ',+' ;
       END IF
       END MACRO;
```

The strings PARM1 and PARM2 will only be transformed just before or during the period in which the EXPRESSION macro immediately controls the scan, and then only at an activation point as defined by EXPRESSION. With the EXPRESSION macro now redefined, the trigger macro defined in (2.20) would convert input text as shown in the following example.

```
POLISH PARM1+PARM2;       ->    123,321,+;
POLISH PARM1*(PARM2+A);   ->    123,321,,+,*;     (10.2)
POLISH A+B*PARM1;         ->    A,B,PARM1,*,+;
```

Note that PARM1 in the third example was not converted because the TERM macro, and not EXPRESSION, was in control of the scan when PARM1 was encountered. The activation points for EXPRESSION, TERM, and FACTOR in the third line of (10.2) can be indicated as follows:

```
EXPR:      POLISH A+B*PARM1;
                  |||
TERM:      POLISH A+B*PARM1;                    (10.3)
                  | |||
FACTOR:    POLISH A+B*PARM1;
                  | | |
```

The trigger macros, being local to EXPR, could only be invoked at the beginning of each term in the expressions and at an addition operator. A more desirable result could perhaps have been achieved by making the trigger macros local to FACTOR or TERM.

Local trigger macros may have trigger macros defined local to themselves. Syntax macros are always global and their definitions should not be nested within other macro definitions. If this is done a diagnostic will be issued by the processor although the syntax macro will still be correctly compiled.

Note that the definition of the local trigger macro must be recognized by whatever macro defining macro is being used during the scan of another macro definition. This means that the local trigger macro definition should start at an activation point within the containing macro definition. While it is possible to have several different macros for recognizing and processing various styles of macro definitions and passing them to the macro compilers, a local trigger macro definition and the definition of its containing macro must be processed by the same macro if they are to be linked together properly.

In addition to making trigger macro invocation more context dependent, the use of local trigger macros can allow the processor to run more efficiently because the it normally will not attempt to match a local trigger macro against the input nearly as often as it will a global trigger macro.

10.2 Macro Definition Macros

The bootstrap trigger macro 'MACBOOT', which is supplied with the processor, will match a base language macro definition if it is preceded by the trigger string 'MACBOOT'. The MACBOOT macro in compiled form is contained by a data statement in the processor. If it were to be coded in base language form it would be written

```
MACBOOT TRIGGER: 'MACBOOT'
   MACDEF:<<'TRIGGER'|'SYNTAX'>
   <0,¬<'MEND;'> ATOMS ';'> 'MEND;'>;          (10.4)
   ANSWER MACDEF; GLOBAL(1)=1; MEND;
```

Note that the activation points in macro definitions processed by MACBOOT are at the boundaries between their statements, at the front of any labels that might exist. Because of this the MEND statement in a macro definition processed by MACBOOT must not be labelled if it is to be recognized and terminate the scan of the MACBOOT macro properly. As described earlier, the MACBOOT macro is rarely used except to define an enhanced macro definition language. A number of base language macro definitions can be found in appendix B, which contains a listing of the standard macro language extension macros. A simpler set of extension macros, using scanning techniques that are perhaps inferior to those used by the standard set, illustrates the macro definition language enhanced by IF-ELSE, FOR, and GO TO statements. The use of the brackets in this example as a DO-END pair has been mentioned before in the section on the input reader.

```
MACBOOT SYNTAX:EXP1
    ID ( '('<1,EXP3/',','>')' ) | NUM | '('EXP6')' | STR ;
        ANSWER MATCH; MEND;
MACBOOT SYNTAX:EXP3  <1,EXP1/ '*'|'/'|'+'|'-' >;
    ANSWER MATCH; MEND;
MACBOOT SYNTAX:EXP4 EXP3 ( RELOP EXP3 ) ;
    ANSWER MATCH; MEND;
MACBOOT SYNTAX:EXP6  <1, ('¬') EXP4/'&'|'|'>;
    ANSWER MATCH; MEND;

MACBOOT SYNTAX:RELOP
    '=' | '¬=' | '>=' | '>' | '<=' | '<' ;
    ANSWER MATCH; MEND;

MACBOOT TRIGGER: 'MACRO' TYP:<'TRIGGER:'|'SYNTAX:'> (ID)
    '<' ATOMS ';' STMENT '>' ;
    ANSWER TYP ID A1 ';' STMENT '; MEND;' ;
    GLOBAL(1)=1;
    MEND;

MACBOOT SYNTAX: STMENT <0,<0, ID ':'> ¬<'>'> TEXT ';'> ;

    MACBOOT TRIGGER: 'FOR' ID '=' EXP3
          ('TO' TO:EXP3)('BY' BY:EXP3) '<' STMENT '>';
          STRING START,NEXT,S;
          GLOBAL(21)=GLOBAL(21)+10; I=GLOBAL(21);
          START='L'||CS(I);
          NEXT='L'||CS(I+1);
          RESCAN ID '=' EXP3 '; IF(1)' START ';';
          IF(BY)LAB1;
          BY='1';
    LAB1: RESCAN NEXT.':' ID '=' ID '+' BY ';' START.':;';
          IF(¬TO) LAB3;
          RESCAN 'IF(' ID '>' TO ') L'.CS(I+2) ';';
    LAB3: RESCAN STMENT ' IF(1)' NEXT '; L'.CS(I+2).':;' ;
          MEND;

    MACBOOT TRIGGER: 'IF'  EXP6 '<' STMENT '>'
        ELSE:('ELSE<' M2:STMENT '>' ) ;
        GLOBAL(21)=GLOBAL(21)+10;  I=GLOBAL(21);
        RESCAN 'IF(¬(' EXP6 ')) L'.CS(I) ';' STMENT ;
        IF(¬ELSE) IF1;
        RESCAN 'IF(1) L'.CS(I+1)';' ;
    IF1:RESCAN 'L'.CS(I).':;' ;
        IF(¬ELSE) IF2;
        RESCAN M2 'L'.CS(I+1).':;' ;
    IF2:;
        MEND;

    MACBOOT TRIGGER: 'GO TO' ID ';' ;                        (10.5)
        RESCAN 'IF(1)' ID ';'; MEND;
    ANSWER MATCH; MEND;
```

The macro whose trigger is MACRO will be the new macro defining macro. The extended statements are all implemented by trigger macros local to the STMENT syntax macro. STMENT will match a sequence of macro definition statements which are loosely defined to be anything followed by a semicolon. STMENT will also specifically match any preceding statement labels so that the activation points in the macro definition being processed are at the beginning of each label and at the beginning of each statement. The groups of statements that STMENT is to match will always be delimited by left and right bracket symbols, so it is necessary that the outermost loop in the STMENT macro be terminated when a right bracket is encountered.

Two of the more complex language extension trigger macros are those used to implement the FOR and IF-ELSE statements. Each of these use the GLOBAL(21) array element to generate unique statement labels. The IF-ELSE and FOR statements nested within other such statements are processed recursively before their containing constructs are.

10.3 The Protect Option

A macro defined with the protect option specified will cause trigger macro invocation to be suppressed from the time that the macro gains control of the scan up until its replacement procedure finally terminates. This means that trigger macro activity is inhibited during the matching of the protected macro and during the matching of any syntax macros or their descendants to any level which are called from it. A macro is given a protection level of one or two by coding the string (P=1) or (P=2) just before its pattern. Thus the first of the following three macros

```
MACBOOT SYNTAX:EXPRESSION (P=2)
   TERM '+' EXPRESSION | TERM2:TERM ;
   etc.

MACBOOT SYNTAX:TERM              (10.7)
   FACTOR '*' TERM | FACT2:FACTOR ;
   etc.

MACBOOT SYNTAX:FACTOR
   '(' EXPRESSION ')' | ID | NUM ;
   etc.
```

is defined to have a protection level of two, which means that all trigger macro activity is suppressed while EXPRESSION, TERM, or FACTOR (assuming the latter two are called from EXPRESSION) are being matched. If (P=1) is instead coded, then all global trigger macros would be suppressed, but any trigger macros local to EXPRESSION, TERM, FACTOR, or other trigger or syntax macros activated during the matching process for EXPRESSION are allowed to be invoked as usual. Note that example (5.12) was coded with P equal to two so that trigger macros could not be invoked in the middle of matching for an identifier. Actually, P equal to one would be sufficient since there are no trigger macros local to ID, DIGIT, or LETTER in (5.12).

The protect level is not additive. Thus if a macro defined to have protection level one calls a syntax macro also defined with protection level one, the protection level is still one, and not two. The maximum of the protection levels of all active macros is the one used, so that if two macros are active, one of which is level one and the other level two, the protection level is two regardless of which macro called the other.

11.0 Output Processor

When no macros are in control and no trigger macros will match the text beginning at the input scan pointer, the output processor is called. Starting at the input scan pointer, this routine will write into the code file one or more characters and then update the input scan pointer to point to the first character that was not written. The number of characters written is defined below. The processor will then begin again its attempts to match trigger macros against the input stream which now begins at a new location. Those characters written into the code file are no longer accessible to the processor.

11.1 Text Atomization

The number of characters written by the output processor is always determined by the following three rules which are applied in the order written:

1. If any leading blanks are present they are written and then the first non-blank character is written.
2. The character just written is examined and one of the following rules is applied.
   a. If the character just written is an apostrophe, then the entire quoted string including the trailing apostrophe is written.
   b. If the character just written is a letter or a digit, all immediately following letters and digits are written.
   c. If a delimiter, excluding apostrophe and blank, has just been written no more non-blank characters will be written.
3. Any trailing blanks present are written and the input scan pointer is set to the next character.

Thus a form of text atomization is present when no macros are in control, so that the outermost trigger macro will not begin matching in the middle of an identifier or quoted string. Thus "atoms" as defined by the output processor are (1) any quoted string, (2) any contiguous string of alphanumeric characters, and (3) any delimiter, except an apostrophe or a blank.

The amount of atomization present when macros are active is defined by the macros themselves, since trigger macro activation can only then take place just before a pattern string or syntax macro call becomes active. For example, the `STR`, `DEL`, `ID`, and `NUM` primitive macros and normal pattern strings will always release control with the input scan pointer on an atom boundary. The `CHAR` primitive or a pattern string followed by an asterisk might not do this, however.

11.2 Output Format

A general output processor and a FORTRAN output processor are currently available. The general processor allows the TAB and NEWLINE directives to be used in `OUTPUT` statement types four to ten. Normally text is written in columns one to 72 as received from the main processor. A string of less than forty contiguous alphanumeric characters not contained within a quoted string will, however, not be broken at a line boundary but instead will be moved to the beginning of the next line. NEWLINE simply causes any following text to begin on a new line. If that text would have begun on a new line anyway, no action is taken. TAB must always have a numeric argument ranging from one to 72. TAB will cause the text which follows to be started in the specified column. If that column has already been passed one blank will be inserted before the following text is written. Note that a dot (.) appearing between a TAB directive and a string element that is to be written will have no affect on the output, thus

```
OUTPUT  TAB(40).'ABC';
OUTPUT  TAB(40)  'ABC';             (11.1)
```

are identical and will both cause the string 'ABC' to appear starting in column forty. The length of each record produced by the output processor is eighty characters. Presently the last eight characters will always be blanks.

TAB and NEWLINE directives are interpreted by the output processor, not the matcher or replacement procedure interpreter. Because of this the directives are not allowed in ANSWER and RESCAN statements: they cause internal instructions to be inserted into text that the matcher and interpreter can not presently handle. While this seems a desirable restriction at the moment, it may prove otherwise later. It is straightforward to allow format control items in any text strings handled by the main processor. Except for being matched by specific primitive syntax macros they would be ignored but passed on by the matcher and replacement procedure interpreter.

While the general output processor is quite capable of producing code formatted according to FORTRAN rules, it is something of a burden to do so. For this reason a FORTRAN output processor is available. The text processed by this routine is converted to FORTRAN statements by interpreting semicolons as end of statement indicators and adding continuation markers where necessary. Any decimal digits which are the first characters of one of these statements are assumed to compose the label for that statement and so will begin in column one. The remainder of that statement and any statements not beginning with digits will begin in column seven. The length of each record in the code file is 80 characters.

12.0 Techniques for Macro Definition and Use

12.1 Four Ways to Implement Language Structure Extension

The following examples show different ways in which the same extended statement can be added to the FORTRAN like base language, and should illustrate some of the advantages and pitfalls of each. An extended IF statement is to be added to the base language in order to allow more than one statement to be executed if the argument of the IF is true. The syntax of this statement will be

```
            'IF LOGEXP '<' <1, STATEMENT> '>'          (12.1)
```

where LOGEXP and STATEMENT match and return FORTRAN logical expressions and statements, respectively. The bracket symbols are used as a DO-END group, as described earlier. If STATEMENT matches only statements of the FORTRAN base language, and the GLOBAL(21) array element is used to generate unique statement label numbers, then a trigger macro to convert (12.1) can be written

```
MACRO TRIGGER:
     'IF' LOGEXP '<' GROUP:<1, STATEMENT> '>';
     LABEL=GLOBAL(21)+10; GLOBAL(21)=LABEL;
     RESCAN 'IF(.NOT.(' LOGEXP '))GO TO' CS(LABEL) ';'
                GROUP                               (12.2)
           CS(LABEL) 'CONTINUE;';
     END MACRO;
```

An extended IF statement which is nested within another extended IF statement will be recursively converted into a string of FORTRAN statements, so that the subpattern labelled by GROUP will always see base language statements. Problems may arise, however, when large members of statements are contained by an IF construct or many IF statements are nested within each other. The processor must retain all statements scanned by the IF macro's loop construct in its internal storage areas. This may not be possible if one IF construct encompasses a great number of statements. In addition, at each level of recursion, all statements scanned by the above macro must be placed back into the input where they will be scanned again, so that a statement nested within several IF constructs must be scanned and recognized a number of times equal to its nesting level. In a language with several such extended constructs, such as the extended macro definition language, this continual rescanning could lead to serious inefficiencies. If these difficulties are not significant in a particular application, however, the above macro should work satisfactorily. Example (10.5) in 10.0, “Macro Definition” on page 63 is an example of a set of macros which use some of the techniques illustrated above to extend the macro definition language. The basic conclusion from this example is that trigger macros which do RESCANs should not be used to match constructs which might contain large amounts of text, or which might be nested within each other to a high degree.

The next syntax macro provides a second way of translating (12.1).

```
MACRO SYNTAX: IF_STATE
      'IF' LOGEXP '<' GROUP:<1, STATEMENT> '>';
      LABEL=GLOBAL(21)+10; GLOBAL(21)=LABEL;
      ANSWER 'IF(.NOT.(' LOGEXP '))GO TO' CS(LABEL) ';'
                  GROUP                                    (11.3)
             CS(LABEL) 'CONTINUE;';
      END MACRO;
```

This macro requires the existence of a calling macro, which, if the extended IF statement is to be considered a statement in the new language, must be the same macro that IF_STATE calls, namely STATEMENT. Whereas before STATEMENT matched statements in the base language, it must now match those of the extended language. STATEMENT must ANSWER text back when called by IF_STATE if the latter is to work properly. Since STATEMENT also calls IF_STATE, it appears that if the type of text matching done by the above macro is consistently employed then each macro will ANSWER the text it produces to the macro that called it. The buck must stop somewhere, of course, and in this case a convenient place might be a trigger macro which recognizes an entire program unit, such as a subroutine. Such a macro might be called by the appearance of the string SUBROUTINE in the input. The SUBROUTINE macro would probably OUTPUT the translated source into the code file. Extended IF statements, or similarly defined constructs, can still be nested within each other since STATEMENT, which is called by IF_STATE, will recognize and translate an extended IF statement. This time, however, each statement is scanned only once, no matter now many extended language constructs that it is nested within.

The standard replacement procedure language extension macros in appendix A use the above method to convert macro definitions into the base language, since the final results must be retained within the processor. The trigger macro which originates the matching process passes the converted text directly to the macro compilers by executing an ANSWER statement and setting GLOBAL(1) equal to one. To summarize the second example, the problem of continually rescanning statements nested within extended language constructs is solved, but the problems that might arise from extended constructs containing large amounts of text remain.

In order to relieve the processor's internal buffers from having to store large quantities of text contained by an extended language construct, the macro defining such a construct can be divided into two macros.

```
MACRO TRIGGER:
    'IF' LOGEXP '<';
    LABEL=GLOBAL(21)+10; GLOBAL(21)=LABEL;
    STACK_POINT=GLOBAL(22)+1; GLOBAL(22)=STACK_POINT;
    IF STACK_POINT>100 THEN
        WARN 'LABEL STACK OVERFLOW';
    ELSE
        GLOBAL(STACK_POINT)=LABEl;
    END IF
    RESCAN 'IF(.NOT.(' LOGEXP '))GO TO' CS(LABEL) ';';
    END MACRO;                                         (11.4)
```

```
MACRO TRIGGER:
    '>';
    STACK_POINT=GLOBAL(22); GLOBAL(22)=STACK_POINT-1;
    IF STACK_POINT<50 THEN
        WARN 'LABEL STACK UNDERFLOW';
    ELSEIF STACK_POINT<=100 THEN
        RESCAN CS(GLOBAL(STACK_POINT)) 'CONTINUE;';
    END IF
    END MACRO;
```

Elements fifty to one hundred of the GLOBAL array are being used as a stack to keep track of statement labels. The opening portion of an extended IF statement generates a branch to a label and stacks that label. The closing portion, which is simply a right bracket, unstacks the appropriate label and generates a target statement for the previous branch. If several extended language constructs make use of the bracket symbols as DO-END pairs then a second number describing the statement type must also be stacked, but the above macros illustrate the basic idea being discussed. Label numbers were effectively stacked for the previous two examples, but this was done automatically as part of the STEP processor's normal recursive operation, and the user did not have to concern himself with it. The above macros then solve both of the problems discussed earlier. Very little space for storing text is required of the processor because only small portions of extended language statements are recognized, converted, and returned to the input from where they will be passed into the code file. Statements of the base language need not be recognized by any macros at all, and so can be immediately passed to the code file when first encountered. Nested constructs will be handled correctly as long as the stack does not overflow.

There are several disadvantages to the above method, however. First of all, macro writing becomes more difficult and the resulting macros are harder to read because the macro must explicitly do work previously done by the processor. If nothing else, it is at least aesthetically unpleasant to have to write more than one macro to handle a single extended language construct. This problem becomes more apparent if an IF-ELSEIF-ELSE-END IF construct or a CASE statement must be implemented in this fashion. It is also unfortunate that the right bracket becomes a reserved symbol: any time a right bracket is encountered, whether in the proper context or not, a label is unstacked and a statement is generated. Assuming that right brackets are used in the proper context, what happens if a trigger macro X, which belongs to some other part of the language extension macro set, becomes active and scans some text which includes a right bracket, and then fails at some later point? When the right bracket is encountered during the scan of macro X, the appropriate trigger macro is recursively activated and a label is unstacked and used to generate a CONTINUE statement. When X finally fails, the CONTINUE statement will disappear along with any other ANSWER or RESCAN text produced during the matching of X. The unstacking of the label is, however, a permanent effect and is not undone. Since the translation process has no way of recovering from such a disaster, the only safe environment in which to use the above macros is one in which no recursive trigger macro operation is allowed. This is easily achieved by coding all macros with a protection level of two.

A STATEMENT macro, which is necessary to the operation of the preceding two examples, is not needed for the present one. In fact, it would be awkward to add macros to recognize all possible statements (including incorrect ones) to this scheme. In some ways this could be an advantage: macros need only be written for constructs that must be converted, others will simply be passed on the the base language compiler. This, of course, means that many of the syntax errors that might exist will be caught by the base language compiler and so will not be related to the original source.

The original problems of processor storage space and continual rescanning of nested statements are not present in this third example, but enough new problems and restrictions on processor operation have been added to make this method unattractive for many applications.

Perhaps the most satisfactory way in which to implement the extended If statement is given by the macro

```
MACRO SYNTAX: IF_STATE
    'IF' LOGEXP '<';
    LABEL=GLOBAL(21)+10; GLOBAL(21)=LABEL;
    OUTPUT 'IF(.NOT.(' LOGEXP '))GO TO' CS(LABEL) ';';
    SCAN <0, STATEMENT>;                           (11.5)
    SCAN BRACKET: '>';
    OUTPUT CS(LABEL) 'CONTINUE;';
    END MACRO;
```

The STATEMENT macro would OUTPUT the text that it scans immediately, so that text will not build up in the processor. Some of the advantages of the above macro are illustrated in the sections on the SCAN and OUTPUT statements. The full control of the scanning process makes it easy to check for various types of errors. For example, if STATEMENT is similar to the EXEC_STATE macro of (8.20), and if the user of these macros forgot to code one or more closing brackets (a common error when using bracket notation), then the loop over STATEMENT in the above example might be terminated by encountering an END statement. Note that, without even trying, the above macro will correct this error. It would be desirable, however, to add

```
IF ¬BRACKET THEN
    WARN '**MISSING RIGHT BRACKET INSERTED**';     (11.6)
END IF
```

after the last SCAN statement in (11.5).

12.2 Beware of Short Triggers

The author once coded five trigger macros which were to match extended FORTRAN statements. In order to insure that these macros could only start to match at the beginning of a statement a trigger string containing only a semicolon was used to match the terminating semicolon of the previous statement. The form of each of the five macros was similar to

```
MACBOOT TRIGGER: ';' (NUM) 'PATTERN STRING' ...    (11.7)
         etc.
```

The optional NUM syntax macro call would match a statement label, if any. Other macros which would at times place null statements consisting of only a semicolon into the input were also present. Events had conspired to place four consecutive semicolons into the input in front of a macro definition, and when the scan pointer reached the first of these, the last of the five short trigger macros to be defined was invoked. After the trigger of this macro matched the first of the four semicolons in the input, and before the NUM syntax could be called, the second semicolon caused the same trigger macro to be invoked recursively. The third and fourth semicolons caused two more copies of the same trigger macro to be invoked, after which the macro definition was found which caused the trigger macro which handled that to be invoked. At this point the macro calls were nested five deep. Finally, after the macro compilers were finished, the scan pointer was beyond the macro definition and the most recently invoked copy of the trigger macro which was called by the fourth semicolon resumed to match the input and quickly failed. The input scan pointer then backed up to the fourth semicolon and one of the other macros having only a semicolon in its trigger was invoked. Again, as this macro attempted to match the macro definition was encountered and compiled again. The trigger macro then failed as before and the scan pointer backed up, etc. Finally, when the author looked at the listing, he found the compiler listing for the same macro appearing over and over again until the job exceeded the operating system's line count and was terminated. After a moderate amount of time spent tearing his hair out trying to find a loop in the processor, the author found that the combination of input and macros present at the time of the "loop" would cause the macro definition to appear 625 (the number of macros having a semicolon for a trigger raised to the power of the number of consecutive semicolons in the input) times.

Had it not been for the effects of macro definition, the processor would have been able to complete its translation of the input correctly although it would have worked harder than necessary to do it.

13.0 Conclusion

It is probable that this preliminary version of the STEP processor will undergo a great deal of change in the near future. Many additions are contemplated, some of which are in various stages of completion. Comments and questions about the processor and any of the projected changes listed below would be appreciated.

13.1 Changes for the Future

It should be possible to develop macros to aid the user in developing custom primitive syntax macros at processor generation time. These could have a significant effect on processor efficiency under appropriate circumstances.

The STEP system performs only a single pass on the source being processed. It should be straightforward to add appropriate control statements to allow multi-pass processing. Since different sets of macros would probably be used for each pass, a method of reading macro object code directly into the macro storage areas from a file might save execution time.

An INCLUDE facility is necessary before the processor can be thought of as being complete.

Should it ever prove desirable, it would be straightforward to introduce a new type of procedure which would have the same form as a replacement procedure, but would be called explicitly from other replacement procedures instead of being activated by a successful pattern match. These would serve as external subroutines for the replacement procedures and could be passed arguments.

13.2 References

1. Leavenworth, B. M., Syntax Macros and Extended Translation. Comm ACM (Nov. 1966), Vol. 3. pp. 790-793.
2. Gries, David, Compiler Construction for Digital Computers, p. 93, John Wiley and Sons, NY.
3. Cook, A. J. and Shustek, L. J., A user's Guide to MORTRAN2. CGTM No. 165, Stanford Linear Accelerator Center, Menlo Park, California.
4. IBM. PL/I Language Point 2257 proposal.
5. Kernighan, Brian W., RATFOR - A Preprocessor for a Rational Fortran, Software-Practice and Experience, Vol. 5, 395-406 (1975).
6. The information on IFTRAN was obtained from a brochure by the General Research Corp. in Santa Barbara, California.
7. Gries, p. 97.

Appendix A

A group of base language STEP macros are given which match a base language STEP macro definition and any components thereof. A macro definition is syntactically correct if and only if it can be matched by the MACDEF syntax macro which is given below. These macros as written all execute an ANSWER MATCH (OUTPUT(1) MATCH) statement, and so only perform the task of syntax recognition. It should be possible to define a set of macros to completely process macro definitions into the object form used internally by STEP, thus eliminating the need for the pattern and replacement compilers. A large amount of macro object code would have to be inserted into the system somehow to start the process, though.

```
          "MATCH A BASE LANGUAGE MACBOOT DEFINITION"
MACBOOT SYNTAX: MACDEF MACTYP (PROTCT) PATTRN ';' REPLAC ;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: MACTYP 'TRIGGER:'|'SYNTAX:' ID;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: PROTCT '(P=' <'0'|'1'|'2'> ')' ;
     OUTPUT(1) MATCH; MEND;

          "MATCH A STANDARD MACBOOT PATTERN DEFINITION"
MACBOOT SYNTAX: PATTRN
     <1, < (PATLAB':') MATITM | '¬<'MATITM'>' > / ' ' | '|' > ;
     OUTPUT(1) MATCH; MEND;

         "MATCH A MATCH ITEM IN A PATTERN DEFINITION"
MACBOOT SYNTAX: MATITM
     PATSTR | SYNCAL | '('PATTRN')' | '<'PATTRN'>'
     | '<'NUM ',' PATTRN ('/' PATTRN) '>' ;
     OUTPUT(1) MATCH; MEND;

       "MATCH A PATTERN STRING IN A PATTERN DEFINITION"
MACBOOT SYNTAX: PATSTR
     STR ('*') ;
     OUTPUT(1) MATCH;
     MEND;

                  "MATCH A SYNTAX MACRO CALL"
MACBOOT SYNTAX: SYNCAL
     <'ID'|'NUM'|'DEL'|'STR'|'CHAR'
      | <'BAL'|'ATOMS'|'TEXT'> <PATSTR| '<' <1,PATSTR/'|'> '>' >
      | ID ;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: PATLAB
     ID;
     OUTPUT(1) MATCH; MEND;

  "MATCH A REPLACEMENT PROCEDURE PORTION OF A MACRO DEFINITION"
MACBOOT SYNTAX: REPLAC
     <0, <0, LABEL ':'> MSTATE > 'MEND;';
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: LABEL  ID ;
     OUTPUT(1) MATCH; MEND;

      "MATCH A REPLACEMENT PROCEDURE STATEMENT EXCEPT MEND"
MACBOOT SYNTAX: MSTATE
     STRING | SYMBOL | ASSIGN | IF | OUTPUT | SCAN | FAIL
     | RETURN | NULL ;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: STRING  'STRING' <1, ID / ',' > ';' ;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: SYMBOL
    'SYMBOL' <1, ('STRING') ID '(*' (',' NUM) ')'/','>';';
    OUTPUT(1) MATCH; MEND;
```

```
MACBOOT SYNTAX: ASSIGN
     < INTVAR | 'GLOBAL('LOGEXP')' > '=' LOGEXP ';'
     | < STRVAR | MATVAR > '=' STREXP ';';
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: IF  'IF(' < LOGEXP | MATVAR > ')' LABEL ';';
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: OUTPUT
    'OUTPUT' ('(' INTEXP ')') <1,<' '|'.'>STRELM> ';';
    OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: MERGE
    'MERGE' <1,LOGEXP / ','> ';';
    OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: SCAN
    ('USING' STREXP) 'SCAN' PATTERN ';';
    OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: MKDP
    <'MARK' | 'DROP'> ID ';';
    OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: FAIL  'FAIL;';
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: RETURN  'RETURN;';
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: NULL  ';' ;
     OUTPUT(1) MATCH; MEND;

                  "MATCH AN INTEGER EXPRESSION"
MACBOOT SYNTAX: LOGEXP  <1, LOGTERM / '|' > ;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: LOGTERM <1, LOGELEMENT / '&'>;
    OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: LOGELEMENT
           ('¬') <RELATION | INTEXP | MATREF>
    OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: RELATION
    <1, INTEXP / RELOP> | STREXP R2:RELOP S2:STREXP ;
    OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: INTEXP  ('+'|'-') <1, TERM / '+' | '-' >;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX:TERM  <1, FACTOR / '*' | '/' >;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX:FACTOR
     'GLOBAL(' LOGEXP ')' | INTFCN | INTVAR | SYMREF
         | SYMPNTR | '('LOGEXP')' ;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX:RELOP
    '=' | '¬=' | '<=' | '<' | '>=' | '>' ;
    OUTPUT(1) MATCH; MEND;

       "MATCH ANY FUNCTION WHICH RETURNS AN INTEGER VALUE"
MACBOOT SYNTAX: INTFCN   'INDEX(' STREXP ',' STREXP ')'
          | 'LEN(' STREXP ')'
          | 'CN(' STREXP ')'
          | ID '(*' <0, ',' LOGEXP > ')'
          | 'SCANOK' ;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: INTVAR  ID;
     OUTPUT(1) MATCH; MEND;
```

```
                  ″MATCH ANY STRING EXPRESSION″
MACBOOT SYNTAX:STREXP  <1, STRELM / '||' >;
     OUTPUT(1) MATCH; MEND;

                   ″MATCH ANY STRING ELEMENT″
MACBOOT SYNTAX: STRELM
    STRFCN  | MATVAR | STRVAR | SYMREF | '('STREXP')';
     OUTPUT(1) MATCH; MEND;

           ″MATCH ANY FUNCTION WHICH RETURNS A STRING″
MACBOOT SYNTAX: STRFCN  'SUBS(' STREXP ',' LOGEXP (',' LOGEXP) ')'
          | 'CS(' LOGEXP ')'
          | MATCH | SOURCE ;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: MATVAR  ID ( '(' <1, LOGEXP / ',' > ')' ) ;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: STRVAR  ID;
     OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: SYMREF
    ID '(' <'@' STREXP | LOGEXP | SYMPNTR> (',' LOGEXP) ')' ;
    OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: SYMPNTR
    ID <('.NONEW')('.LOCAL') | ('.LOCAL')('.NONEW')> '@' STREXP;
    OUTPUT(1) MATCH; MEND;
```

Appendix B: Standard Macro Language Extensions

The standard macro definition language extension macro set does not completely parse and check the base language, it only converts extended language constructs into base language counterparts. The following macros only represent a few day's work to code and debug, and so can be easily changed if required.

The 21st element of the GLOBAL array is used to generate unique statement labels. The 23rd element is set to a number which indicates (for error checking purposes only) the type of extended construct that the scan pointer is immediately nested within. Its value is one for the LOOP and WHILE constructs, two for the FOR construct, and three for the IF-ELSEIF-ELSE construct. The 22nd element of the GLOBAL array is used to check for errors when processing NEXT and EXIT statements. If these statements do not appear nested (not necessarily immediately) within a FOR, WHILE, or LOOP construct an error is flagged. Note that DeMorgan's rules are fully applied to any logical expression which must be negated.

```
  62   9  10  37
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_$ ,.+-*/()=;:'″#@?|&¬<>!%
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_$ ,.+-*/()=;:'″#@?|&¬<>!%''
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_$ ,.+-*/()=;:'″#@?|&¬<>!%''
   6   9   6   1   2   3   4   7   8

″***********                                ***********″
″***********        THE STEP PROCESSOR      ***********″
″***********                                ***********″
″***********        BY JACK W. SIMPSON      ***********″
″***********    COMPUTATION RESEARCH GROUP  ***********″
″*********** STANFORD LINEAR ACCELERATOR CENTER ***********″
″***********      MENLO PARK, CALIFORNIA    ***********″
″***********                                ***********″
″***********          PRINTED IN USA        ***********″
″***********                                ***********″

MACBOOT TRIGGR:
      'MACRO' TYPE:ID. ':' (ID) ;
      GLOBAL(23)=0; GLOBAL(22)=0;
      SCAN GROUP:<1,MSTMNT> 'END MACRO';
      OUTPUT(1) TYPE ':' ID GROUP ';MEND;';
      GLOBAL(1)=1; MEND;

MACBOOT SYNTAX: MSTMNT
      LAB:<0,ID ':'>
      <'IF' IFSTMT    |
      'FOR' FORSTM    |
      'WHILE' WHILST  |
      'LOOP' LOOPST   |
      'ANSWER' ANSTAT |
      'RESCAN' RESTAT |
      'OUTPUT' OUTSTA |
      'WARN'  WRSTAT  |
      OTHER:<(KEY:ID.) (TYPE:ID.) ATOMS ';'>> ;

      IF (KEY='ELSEIF' | KEY='ELSE') IFSTAT;
      IF (KEY='END') MACEND;
      OUTPUT(1) LAB IFSTMT FORSTM WHILST LOOPST
                ANSTAT RESTAT OUTSTA WRSTAT OTHER;
      RETURN;
IFSTAT: IF (GLOBAL(23)=3) MFAIL;
      OUTPUT(5) 'MISPLACED  ELSE/ELSEIF  DELETED';
      OUTPUT(2) TYPE ATOMS ';' ; RETURN;
MACEND: IF (GLOBAL(23)¬=0 | TYPE='MACRO') MFAIL;
      OUTPUT(5) '″'.KEY TYPE.'″ DELETED - NOTHING TO CLOSE';
      OUTPUT(2) ATOMS ';'; RETURN;
MFAIL: FAIL;
```

```
   MACBOOT TRIGGR:
         'NEXT;';
         I=GLOBAL(22);
         IF (I=0) ERROR;
         OUTPUT(2) 'IF(1) L'.CS(I+1) ';' ;
         RETURN;
ERROR:   OUTPUT(5) '″NEXT″ DELETED - NOT WITHIN LOOP';
         MEND;

   MACBOOT TRIGGR:
         'EXIT;';
         I=GLOBAL(22);
         IF (I=0) ERROR;
         OUTPUT(2) 'IF(1) L'.CS(I+2) ';' ;
         RETURN;
ERROR:   OUTPUT(5) '″EXIT″ DELETED - NOT WITHIN LOOP';
         MEND;

   MACBOOT TRIGGR:
         'GO TO' ID ';' ;
         OUTPUT(2) 'IF(1)' ID ';';
         MEND;

   MACBOOT TRIGGR:
         'IF' $LEXPR 'THEN' <'GO TO' ID | A:'NEXT' | B:'EXIT'>
                         ';END IF';
         OUTPUT(2) 'IF(' $LEXPR ')' ID;
         IF(ID) MACEND;
         I=GLOBAL(22); IF(I¬=0) MOK; FAIL;
   MOK:  OUTPUT(2) 'L';
         IF(B) BE;
         OUTPUT(2) .CS(I+1);
         IF(1) MACEND;
   BE:   OUTPUT(2) .CS(I+2);
   MACEND: OUTPUT(2) ';'; MEND;

      ;MEND;

MACBOOT SYNTAX: IFSTMT
      $NLEXP 'THEN' | BAL 'THEN' ;
      IF (¬BAL) LAB0;
      OUTPUT(5) 'BAD EXPRESSION IN IF STATEMENT';
LAB0: MSAVE=GLOBAL(23); GLOBAL(23)=3;
      SCAN
          GROUP1:<0,MSTMNT>
      <0, 'ELSEIF' EXP2:$NLEXP 'THEN'
          GROUP2:<0,MS2:MSTMNT>>
      ('ELSE' GROUP3:<0,MS3:MSTMNT>)
      'END' ID ;

      L1=GLOBAL(21)+10; GLOBAL(21)=L1; L2=L1;
      OUTPUT(1) 'IF(' $NLEXP ')L'.CS(L1+1) ';'
                  GROUP1 ;
      I=1; EIFNO=GROUP2(*);
ELSEIF: IF (I>EIFNO) ELSE;
      OUTPUT(1) 'IF(1)L'.CS(L1+2) ';L'.CS(L2+1).':;' ;
      L2=GLOBAL(21)+10; GLOBAL(21)=L2;
      OUTPUT(1) 'IF(' EXP2(I) ')L'.CS(L2+1) ';'
                  GROUP2(I);
      I=I+1; IF(1) ELSEIF;
ELSE: IF (¬GROUP3) ENDIF;
      OUTPUT(1) 'IF(1)L'.CS(L1+2) ';L'.CS(L2+1).':;'
                  GROUP3 ;
ENDIF: IF (GROUP3 | GROUP2(*)¬=0) EXTRA;
      OUTPUT(1) 'L'.CS(L1+1).':;' ; IF(1) CHECK;
EXTRA: OUTPUT(1) 'L'.CS(L1+2).':;' ;
CHECK: IF (ID='IF') MACEND;
      OUTPUT(5) '″END IF″ INSERTED BEFORE ″END' ID.'″' ;
      OUTPUT(2) 'END' ID;
MACEND: GLOBAL(23)=MSAVE; MEND;

MACBOOT SYNTAX: FORSTM
      ID '=' $AEXPR ('TO' TO:$AEXPR)('BY' BY:$AEXPR) 'DO'
            | ATOMS <'DO' | ';'> ;
      IF (ID) LAB0;
      OUTPUT(5) 'INCORRECT "FOR" STATMENT SYNTAX';
LAB0: I=GLOBAL(21)+10; GLOBAL(21)=I;
      LSAVE=GLOBAL(22); GLOBAL(22)=I;
      MSAVE=GLOBAL(23); GLOBAL(23)=2;
      SCAN
      GROUP:<0,MSTMNT> 'END' ENDID:ID. ;

      STRING START, NEXT, LAST;
      START='L'||CS(I); NEXT='L'||CS(I+1); LAST='L'||CS(I+2);
      OUTPUT(1) ID '=' $AEXPR ';IF(1)' START ';' ;
      IF(BY) LAB1;
      BY='1';
LAB1: OUTPUT(1) NEXT.':' ID '=' ID '+' BY ';' START.':;' ;
      IF(¬TO) LAB3;
      OUTPUT(1) 'IF(' ID '>' TO ')' LAST ';' ;
LAB3: OUTPUT(1) GROUP 'IF(1)' NEXT ';' LAST.':;' ;
      GLOBAL(22)=LSAVE; GLOBAL(23)=MSAVE;
      IF (ENDID='FOR') MACEND;
      OUTPUT(5) '"END FOR" INSERTED BEFORE "END' ENDID.'"' ;
      OUTPUT(2) 'END' ENDID;
MACEND:; MEND;

MACBOOT SYNTAX: WHILST
      $NLEXP 'DO' | ATOMS <'DO' | ';'> ;

      IF ($NLEXP) LAB0;
      OUTPUT(5) 'BAD EXPRESSION IN "WHILE" STMNT';
LAB0: I=GLOBAL(21)+10; GLOBAL(21)=I;
      LSAVE=GLOBAL(22); GLOBAL(22)=I;
      MSAVE=GLOBAL(23); GLOBAL(23)=1;

      SCAN
      GROUP:<0,MSTMNT> 'END' ID;

      STRING NEXT, LAST; NEXT='L'||CS(I+1); LAST='L'||CS(I+2);
      OUTPUT(1) NEXT.':IF(' $NLEXP ')' LAST ';'
                        GROUP
                'IF(1)' NEXT ';' LAST.':;';
      GLOBAL(22)=LSAVE; GLOBAL(23)=MSAVE;
      IF (ID='WHILE') LAB1;
      OUTPUT(5) '"END WHILE" INSERTED BEFORE "END' ID.'"' ;
      OUTPUT(2) 'END' ID;
LAB1: ;MEND;

MACBOOT SYNTAX: LOOPST
      CHAR ;
      I=GLOBAL(21)+10; GLOBAL(21)=I;
      LSAVE=GLOBAL(22); GLOBAL(22)=I;
      MSAVE=GLOBAL(23); GLOBAL(23)=1;
      SCAN
      GROUP:<0,MSTMNT> 'END' ID ;

      STRING NEXT, LAST; NEXT='L'||CS(I+1); LAST='L'||CS(I+2);
      OUTPUT(1) NEXT.':;' GROUP 'IF(1)' NEXT ';' LAST.':;' ;
      GLOBAL(22)=LSAVE; GLOBAL(23)=MSAVE;
      IF(ID='LOOP') MACEND;
      OUTPUT(5) '"END LOOP" INSERTED BEFORE "END' ID.'"' ;
      OUTPUT(2) 'END' ID;
MACEND:; MEND;
```

```
MACBOOT SYNTAX: ANSTAT
      ATOMS ';';
      OUTPUT(1) 'OUTPUT(1)' MATCH;
      MEND;

MACBOOT SYNTAX: RESTAT
      ATOMS ';';
      OUTPUT(1) 'OUTPUT(2)' MATCH;
      MEND;

MACBOOT SYNTAX: OUTSTA
      UNIT: ( '(' BAL ')' ) ATOMS ';';
      OUTPUT(1) 'OUTPUT';
      IF(UNIT)LABEL;
      OUTPUT(1) '(4)';
LABEL: OUTPUT(1) MATCH;
      MEND;

MACBOOT SYNTAX: WRSTAT
      ATOMS ';';
      OUTPUT(1) 'OUTPUT(5)' MATCH ;
      MEND;

MACBOOT SYNTAX: $LEXPR
      <1, $OREXP / '|'>;
      OUTPUT(1) MATCH;
      MEND;

MACBOOT SYNTAX: $OREXP
      <1, $LOGIC / '&'>;
      OUTPUT(1) MATCH;
      MEND;

MACBOOT SYNTAX: $LOGIC
      $RELAT | ('¬') $REF | '('$LEXPR')' | '¬('$NLEXP')' ;
      IF ($NLEXP) LABEL;
      OUTPUT(1) MATCH; RETURN;
LABEL: OUTPUT(1) '('$NLEXP')';
      MEND;

MACBOOT SYNTAX: $RELAT
      NOT:('¬') A:$AEXPR $RELOP B:$AEXPR ;
      IF (¬NOT) LABEL;
      $RELOP=SUBS('¬== <=< >=> ',
            INDEX('= ¬=> >=< <=',$RELOP) , 2);
LABEL: IF (A¬='0') NSWTCH;
      STRING S; S=A; A=B; B=S;
NSWTCH:;
      IF (B¬='0') NORMAL;
      IF ($RELOP='¬=') NEQU;
      IF ($RELOP¬='=' & $RELOP¬='= ') NORMAL;
      OUTPUT(1) '¬' ;
NEQU: OUTPUT(1) A ;
      RETURN;
NORMAL: OUTPUT(1) A $RELOP B;
      MEND;

MACBOOT SYNTAX: $AEXPR
      <1,SIMEXP/ '*' | '/' | '+' | '-' | '||' > ;
      OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: SIMEXP
      ¬<'THEN'|'DO'><$REF | NUM | '(' $AEXPR ')' | STR> ;
      OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: $REF
      ID ( '(' < $AEXPR | '*' > <0, ',' $AEXPR > ')' ) ;
      OUTPUT(1) MATCH; MEND;
```

```
MACBOOT SYNTAX: $RELOP
      '=' | '¬=' | '>=' | '>' | '<=' | '<' ;
      OUTPUT(1) MATCH; MEND;

MACBOOT SYNTAX: $NLEXP
      <1, $NOREX / $NOR>;
      OUTPUT(1) MATCH;
      MEND;

MACBOOT SYNTAX: $NOR
      '|';
      OUTPUT(1) '&';
      MEND;

MACBOOT SYNTAX: $NOREX
      <1, $NTINS $LOGIC / $NANOP>;
      IF ($LOGIC(*)>1) LABEL;
      OUTPUT(1) MATCH; RETURN;
LABEL: OUTPUT(1) '(' MATCH ')';
      MEND;

MACBOOT SYNTAX: $NANOP
      '&';
      OUTPUT(1) '|';
      MEND;

MACBOOT SYNTAX: $NTINS
      N:('¬');
      IF (N) LABEL;
      OUTPUT(2) '¬';
LABEL:; MEND;
```

Appendix C: General Examples

Example 1: Constant Propagation in Expressions

The following macro set recognizes FORTRAN integer expressions and folds all constants possible as it does so. Note that constant folding will take place across a division operator only if the division can be performed with no remainder or if no variables appear to the left of the division operator. Local trigger macros operate asynchronously during the expression parse to simplify certain forms.

```
MACRO SYNTAX: ARIEXP
    PREOP:('+'|'-')<1, TERM / INOP:<'+'|'-'>>;

    MACRO TRIGGR:
        '+(' UNARY:('+'|'-') ARIEXP ')' ENDOP:<')'|'+'|'-'>;
        IF ¬UNARY THEN UNARY='+'; END IF
        RESCAN UNARY ARIEXP ENDOP;
        END MACRO;

    MACRO TRIGGR:
        '-(' (PMCONV) ARICON ')' ENDOP:<')'|'+'|'-'>;
        IF ¬PMCONV THEN PMCONV='-'; END IF
        RESCAN PMCONV ARICON ENDOP;
        END MACRO;

    STRING OPERAT,S;
    NTERMS=TERM(*);
    OPERAT=PREOP; SUM=0; ANSFLG=0;

    FOR I=1 TO NTERMS DO
        S=TERM(I);

        "IF A MINUS SIGN WAS APPENDED BY TERM, CHANGE SIGN
        OF OPERAT."
        IF SUBS(S,LENGTH(S),1)='-' THEN
            S=SUBS(S,1,LENGTH(S)-1);
            IF OPERAT='-' THEN
                OPERAT='+';
            ELSE
                OPERAT='-';
            END IF
        END IF

        "USE SCAN TO FIND OUT IF THIS TERM IS A CONSTANT -
        IF IT IS ADD TO TOTAL INSTEAD OF ANSWERING TEXT"
        USING S SCAN NUM;
        IF SCANOK THEN
            SUM=SUM+CN(OPERAT||S);
        ELSE
            IF ANSFLG=1 | OPERAT='-' THEN ANSWER OPERAT; END IF
            ANSFLG=1;
            ANSWER S;
        END IF
        OPERAT=INOP(I);
    END FOR

    "FINALLY, APPEND THE CONSTANT PORTION TO THE REST"
    IF SUM¬=0 | ANSFLG=0 THEN
        IF SUM>0 & ANSFLG=1 THEN ANSWER '+'; END IF
        ANSWER CS(SUM);
    END IF
    END MACRO;
```

```
MACRO SYNTAX: ARICON
    (PMCONV) <1, TERM / PM2:PMCONV>;
    ANSWER MATCH;
    END MACRO;

MACRO SYNTAX: PMCONV
    PLUS:'+' | MINUS:'-';
    IF PLUS THEN
        ANSWER '-';
    ELSE
        ANSWER '+';
    END IF
    END MACRO;

MACRO SYNTAX:TERM
    INSST <1, <1, '*' FACTOR> | '/' DIVIDE:FACTOR>;

    MACRO TRIGGR:
        '/(';
        RESCAN NOTRIG MATCH;
        END MACRO;

    MACRO TRIGGR:
        '(' MINUS:('-') TERM MINUS2:('-') ')' ¬<'**'>;
        USING MATCH SCAN '(' BAL '/' BAL ')';
        IF SCANOK THEN FAIL; END IF
        RESCAN TERM;
        I=1;
        IF MINUS THEN I=-1; END IF
        IF MINUS2 THEN I=-I; END IF
        IF I<0 THEN RESCAN '*(-1)'; END IF
        END MACRO;

    STRING FACSTR,DIVSTR;
    PRODUC=1; ALG=0; SIGN=1;
    LIM2=FACTOR(*);

    "THIS LOOP CORRESPONDS TO THE OUTER LOOP IN THE ABOVE PATTERN"
    "NO MORE THAN ONE DIVISOR IS DEALT WITH DURING EACH ITERATION"
    FOR II=1 TO LIM2 DO

        LIM=FACTOR(*,II);
        FOR I=1 TO LIM DO
            FACSTR=FACTOR(I,II);
            USING FACSTR SCAN NUM | '(-' MNEG:<NUM2:NUM|REF> ')';
            IF MNEG THEN FACSTR=MNEG; SIGN=-SIGN; END IF
            IF SCANOK & ¬REF THEN
                PRODUC=PRODUC*CN(FACSTR);
            ELSE
                IF ALG¬=0 THEN ANSWER '*'; END IF
                ANSWER FACSTR;
                ALG=1;
            END IF
        END FOR

        IF PRODUC=0 THEN EXIT; END IF

        IF DIVIDE(II) THEN
            DIVSTR=DIVIDE(II);
            USING DIVSTR SCAN NUM3:NUM |
                  '(-' DNEG:<NUM4:NUM | REF2:REF> ')';
            IF DNEG THEN DIVSTR=DNEG; SIGN=-SIGN; END IF
            IF SCANOK & ¬REF2 THEN
                DIV=CN(DIVSTR);
                RESULT=PRODUC/DIV;
                IF RESULT*DIV¬=PRODUC & ALG¬=0 THEN
                              GO TO KEEP; END IF
                PRODUC=RESULT;
            ELSE KEEP:
                IF ALG=0 | PRODUC¬=1 THEN
                    IF ALG¬=0 THEN ANSWER '*'; END IF
                    ANSWER CS(PRODUC);
                END IF
                ANSWER '/' DIVSTR;
                PRODUC=1;
            END IF
        END IF
    END FOR

    "FINALLY, APPEND THE CONSTANT PORTION TO THE REST"
    IF PRODUC¬=1 | ALG=0 THEN
       IF ALG¬=0 THEN ANSWER '*'; END IF
       ANSWER CS(PRODUC);
    END IF

    "IF AN ODD NUMBER OF UNARY MINUS SIGNS HAVE BEEN REMOVED FROM"
    "A TERM, INFORM ARIEXP OF THIS FACT"
    IF SIGN<0 THEN ANSWER .'-'; END IF
    END MACRO;

MACRO SYNTAX: INSST   "INSST IS USED IN ORDER THAT EVERY FACTOR"
    CHAR;             "APPEARING IN A TERM WILL BE PRECEDED BY"
    RESCAN '*' CHAR;  "THE APPROPRIATE OPERATOR.  THIS ALLOWS"
    END MACRO;        "THE PATTERN FOR TERM TO BE CONVENIENTLY"
                      "STRUCTURED"
    END MACRO;

MACRO SYNTAX: FACTOR
    <REF | NUM | '(' ARIEXP ')'> EXPON:('**' FACTOR) ;
    IF ARIEXP THEN
        USING ARIEXP SCAN REF2:REF | NUM2:NUM ;
        IF SCANOK THEN ANSWER ARIEXP EXPON; RETURN; END IF
    END IF
    ANSWER MATCH;
    END MACRO;

MACRO SYNTAX: REF
      ID ( '(' <1,ARIEXP/ ',' > ')' ) ;
      ANSWER MATCH;
      END MACRO;

"NOW WRITE A TRIGGER MACRO TO TEST THE EXPRESSION MACROS"

MACRO TRIGGR:
    'STARTEXP' ARIEXP ';';
    RESCAN NOTRIG MATCH;
    END MACRO;

"NOW CONVERT THE EXPRESSIONS IN THE FOLLOWING LINES"

STARTEXP A*(B+C/D);
STARTEXP 2*(3+4);
STARTEXP 2*(A+3-B+4)*9;
STARTEXP A+2-(8+G*C(3-A+4))+G-5;
STARTEXP A*5*3/2*4+B*3*4/6*C;
STARTEXP (A+C*4/2)/8*C+9-3*R*5/1;
```

When this macro was tested, the above input expressions were converted to

```
STARTEXP A * ( B + C / D) ;
STARTEXP 14 ;
STARTEXP ( A - B + 7) * 18 ;
STARTEXP A - G * C( - A + 7) + G -11 ;
STARTEXP A * 15 / 2 * 4 + B * C * 2 ;
STARTEXP ( A + C * 2) / 8 * C - R * 15 + 9 ;
```

Example 2: SNOBOL Pattern Matching Statements in PL/I

The following macro set will implement a SNOBOL-like pattern matching statement as an extended PL/I statement. The macros were transliterated from the IBM Language Point 2257 proposal, example 3. Various utility macros, such as `REF`, `LABEL`, etc. were copied from page 10 of the proposal. The implementation is not identical to that in the proposal because of various errors that were corrected.

```
     The following macros would process the statement

         (SNOBOL): REPEAT: A(I) X '.' Y = '' /S(REPEAT);
```

In this particular statement, `A(I)` represents a character string that is searched for the substring `X||'.'||Y`. If the substring is found and the equal sign followed by a string expression has been coded then the substring is replaced by the string expression on the right side of the equal sign (in this case the substring is removed). If the slash is coded then one or two labels must follow it surrounded by parentheses and preceded by the letter `S` for success or `F` for failure. Control is passed to the appropriate label depending upon the success or failure of the pattern match.

The PL/I code generated from the above statement is

```
  REPEAT: NTEMP$$$=INDEX(A(I), X||'.'||Y);
          IF NTEMP$$$¬=0 THEN DO
              A(I)=SUBSTR(A(I),1,NTEMP$$$) || '' ||
                     SUBSTR(A(I),NTEMP$$$+LENGTH(X||'.'||Y));
              GO TO REPEAT;
          END;
```

where the indentation has been added by the author.

```
MACRO TRIGGR:
    '(SNOBOL):' (PREFIX) (LABLST)
         REF STRNG1 ('=' STRNG2:STRNG1)
         ('/' <1, T:('S'|'F') '('LABEL')'> ) ';';

    STRING STMT1,STMT2; ERROR=0;

    "NOW CHECK THE TARGET LABELS AND GENERATE CODE INTO
    INTO STMT1 FOR SUCCESS LABEL AND STMT2 FOR FAILURE LABEL"

    FOR I=1 TO T(*) DO
       IF T(I)='S' THEN
          IF STMT1>='' THEN
             ERROR=1;
          ELSE
             STMT1=PREFIX||' GO TO '||LABEL(I)||';';
          END IF
       ELSEIF T(I)='F' THEN
          IF STMT2>='' THEN
             ERROR=1;
          ELSE
             STMT2='ELSE'||PREFIX||' GO TO '||LABEL(I)||';';
          END IF
       ELSE
          ERROR=1;
       END IF
    END FOR

    "NOW FIELD ERRORS FOUND WHEN LOOKING FOR TARGET LABELS"

    IF ERROR¬=0 THEN
       WARN 'ERROR IN SNOBOL STATEMENT -- STATEMENT DELETED';
       OUTPUT '/* **** ERROR ****  STATEMENT - ' SOURCE
              '- DELETED */';
       RETURN;
    END IF;

    "OUTPUT THE CODE FOR THE STATEMENT"

    OUTPUT PREFIX LABLST
           'NTEMP$$$=INDEX(' REF ',' STRNG1 ');'
           'IF NTEMP$$$¬=0 THEN DO;';

    IF STRNG2 THEN
       OUTPUT PREFIX
              REF '=SUBSTR(' REF ',1,NTEMP$$$)||'
                  STRNG2 '||SUBSTR(' REF
                  ',NTEMP$$$+LENGTH(' STRNG1 '));';
    END IF
    OUTPUT STMT1 'END;' STMT2;
    END MACRO;

MACRO SYNTAX: STRNG1        "A SNOBOL STYLE STRING EXPRESSION,"
    <1, ITEM: <REF | STR> / ' '>;
    ANSWER ITEM(1);         "WHERE A BLANK INDICATES CONCATENATION"
    FOR I=2 TO ITEM(*) DO   "IS RECOGNIZED AND TRANSLATED INTO ITS"
       ANSWER '||' ITEM(I); "PL/I COUNTERPART."
    END FOR;
    END MACRO;

MACRO SYNTAX:EXPR         "A PL/I EXPRESSION OF NEARLY ANY TYPE"
    <1, PRIMIT / OPERAT>; "WILL BE MATCHED BY THIS AND THE"
    ANSWER MATCH;         "FOLLOWING MACROS"
    END MACRO;

MACRO SYNTAX: PRIMIT
    '(' EXPR ')' | <'¬'|'+'|'-'> EXPR |
    REF | STR | NUM ;
    ANSWER MATCH;
    END MACRO;

MACRO SYNTAX: OPERAT     "THIS MATCHES A PL/I INFIX OPERATOR"
    '**' | '*' | '/' | '+' | '-' | '>=' | '>' | '<=' |
    '<' | '¬=' | '=' | '¬>' | '¬<' | '||' | '&' | '|' ;
    ANSWER MATCH;       "IT IS RETURNED UNALTERED TO THE CALLER"
    END MACRO;

MACRO SYNTAX: REF        "THIS MATCHES A PL/I REFERENCE"
    <1, ID ( '(' <1, EXPR / ','> ')' ) / '->' | '.'>;
    ANSWER MATCH;        "AND RETURNS IT UNALTERED"
    END MACRO;

MACRO SYNTAX: LABLST     "THIS MACRO MATCHES A SEQUENCE OF ONE"
    <1, LABEL ':'>;     "OR MORE PL/I LABELS (IDENTIFIERS)"
    ANSWER MATCH;       "FOLLOWED BY COLONS"
    END MACRO;

MACRO SYNTAX: LABEL      "THIS MACRO MATCHES A SINGLE PL/I LABEL,"
    ID;                 "WHICH IS SIMPLY AN IDENTIFIER.  A MORE"
    ANSWER MATCH;       "COMPLEX MACRO MIGHT ADDITIONALLY CHECK"
    END MACRO;          "THAT THE ID IS REALLY A LABEL BY USING"
                        "THE SYMBOL TABLE FACILITY."

MACRO SYNTAX: PREFIX
    '(' <1, ID / ','> '):';
    ANSWER MATCH;
    END MACRO;
```

Example 3: Structured FORTRAN Preprocessor

The following macro set is quite long, and forms the nucleus of a structured FORTRAN preprocessor. The preprocessor is written with the aid of the macro language extension set given in appendix B. The appearance of the text BEGIN PROCESSOR SCAN in the input will trigger the preprocessor, which will then retain control of the scan until the end of input is reached. Main programs must begin with the identifier PROGRAM. Language extensions implemented include IF-ELSEIF-ELSE, SELECT (a type of case statement), WHILE, UNTIL, LOOP, EXIT, and other statements. The entire FORTRAN base language, including FORMAT statements, is completely parsed and checked for correctness. In addition to this, certain conveniences not recommended for general use have been added according to the whims of the author. For example, one need not code the IF in an IF statement, but simply begin the statement with a left parenthesis. This is implemented by a local trigger macro. Also, to save keystrokes, one may replace the keyword ELSEIF by an asterisk, etc.

At present this preprocessor is primarily concerned with the extension of the control structures of the base language, and for this purpose it would not be necessary to do a complete parse, but only to recognize and convert the extended statements. With a little more work, however, the symbol table facility could be put to use and allow this macro set to be extended so that new data types and structures could be added to the language.

The GLOBAL(31) array element is used to generate unique statement labels, while GLOBAL(32) is used to pass a symbol array pointer from the LABGEN syntax macro, which recognizes statement label declarations, to the immediately following statement. This information is presently needed only for the implementation of the EXIT:label:; and the NEXT:label:; statements. Note that the constant propagation macros ARIEXP, TERM, and FACTOR which appear in a previous example can easily be substituted for their counterparts in this macro set.

This is the preprocessor that is used for the development of STEP itself. To give an idea of the type of language that these macros process, the following routine that performs all of the MARK and DROP functions for the replacement procedures is listed from the STEP source.

```
SUBROUTINE MKDP(IA,IAS,IER); INTEGERALL; CMMNS; SYMCMMNS;

IER=0; IL=AP(IAS+3);

(IA=MRK$)
   < +AP(IAS+2);               "INCREMENT MARK COUNT IN HEADER"
   (IFBLK=0)<ERR('SYMBOL ARRAY PNTR OVERFLOW'); STOP;>
   J=IFBLK; IFBLK=AP(J);     "REMOVE BLOCK FROM FREE BLOCK LIST"
   (AP(J+1)=1)CALL AGARBG(1);"IF DROP MARKED IT, COLLECT GARBAGE"
   AP(J+1)=IL; AP(IAS+3)=J+1; "MARK LINK NEW TO OLD & HEADER TO NEW"
   AP(J)=0; AP(J+3)=0;       "ZERO NEXT BLOCK & TREE ROOT POINTERS"
   J=J+4; AP(J-2)=J;         "LINK FREE LIST HEADER TO 1ST ELEMENT"
   IN=AP(IAS+1); (0>IN) IN=-IN; IN=IN+3; NUM=$($BLKLEN-4)/IN;
   DO I=1,NUM<JJ=J; J=J+IN; AP(JJ)=J;> AP(JJ)=0;>
ELSE
   < -AP(IAS+2);                  "DECREMENT MARK COUNT"
   (IL=0)<ERR('NO DROP ON SYMBOL ARRAY'); RETURN;>
   AP(IAS+3)=AP(IL); J=IL-1;      "DELETE BLOCK FROM MARK LIST"
   WHILE(J¬0)<JJ=J; J=AP(J);      "SAV ADDR OF NEXT BLOCK IN J"
         IN=JJ+$($BLKLEN-1); DO I=JJ,IN<AP(I)=0;>   "CLEAR BLOCKS"
         (IFBLK¬0)<AP(IBBLK)=JJ;> "IF FREE BLOCK LIST EMPTY"
         ELSE <IFBLK=JJ;>         "IF NOT EMPTY"
         IBBLK=JJ; AP(JJ+1)=1;>>  "PNT TO LAST BLK & MARK FOR AGARBG"
RETURN; END;
```

```
"***********                                    ***********"
"***********         THE STEP PROCESSOR         ***********"
"***********                                    ***********"
"***********         BY JACK W. SIMPSON         ***********"
"***********     COMPUTATION RESEARCH GROUP     ***********"
"*********** STANFORD LINEAR ACCELERATOR CENTER ***********"
"***********       MENLO PARK, CALIFORNIA       ***********"
"***********                                    ***********"
"***********           PRINTED IN USA           ***********"
"***********                                    ***********"

"**********  FORTRAN PREPROCESSOR MACRO SET FOLLOWS ********"

MACRO TRIGGR:
      'BEGIN PROCESSOR SCAN'
      <0,< MAINPR | BLOCKD | SUBROU | FUNCTI | SERROR >>;
      WARN '*** PROCESSOR SCAN COMPLETED ***';
      END MACRO;

MACRO SYNTAX: MAINPR
      'PROGRAM' ROUTIN ;
      END MACRO;

MACRO SYNTAX: BLOCKD
      'BLOCK DATA' TERMIN (IMPLIC) ;
      OUTPUT 'BLOCK DATA;' IMPLIC;
      SCAN <0,< DCSTAT | ERRORS>> 'END;' ;
      OUTPUT 'END;' ;
      END MACRO;

MACRO SYNTAX: SUBROU
      'SUBROUTINE' ID ( '(' ARGLST ')' ) TERMIN ;
      OUTPUT MATCH ;
      SCAN ROUTIN ;
      END MACRO;

MACRO SYNTAX: FUNCTI
      (TYPE) 'FUNCTION' ID ( '(' ARGLST ')' ) TERMIN ;
      OUTPUT MATCH ;
      SCAN ROUTIN;
      END MACRO;

MACRO SYNTAX: TERMIN
      T:(';');
      IF ¬T THEN
          WARN '*** MISSING TERMINATOR INSERTED ***';
      END IF
      ANSWER ';';
       END MACRO;

MACRO SYNTAX: RBRACK
      A:('>');
      IF ¬A THEN
          WARN 'MISSING RIGHT BRACKET INSERTED';
      END IF
      END MACRO;

MACRO SYNTAX: TYPE
      'REAL'|'DOUBLE PRECISION'|'INTEGER'|'COMPLEX'|'LOGICAL';
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: SERROR
      ';' | ATOMS ';';
      IF ATOMS THEN
          WARN '*** STRING  ' ATOMS '  -- FOUND BETWEEN PROGRAM UNITS';
      END IF
      END MACRO;
```

```
MACRO SYNTAX: ARGLST
      <1, ID / ',' > <0, ',*'>;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: ROUTIN
      (IMPLIC) <0,DCSTAT> ;
      SYMBOL LABEL(*,2) ;
      GLOBAL(31)=90; GLOBAL(32)=0; GLOBAL(33)=0;
      LOOP
          SCAN SGROUP < '>' | END:'END;'> ;
          IF END THEN EXIT; END IF
          WARN '*** EXTRA RIGHT BRACKET DELETED ***';
      END LOOP
      OUTPUT 'END;' ;
      DROP LABEL;  MARK LABEL;
      END MACRO;

MACRO SYNTAX: SGROUP
      <0, <0, LABGEN> STATEM>;

      MACRO TRIGGR:
          'WHILE(' BAL ')<' ;
          RESCAN 'UNTIL(¬(' BAL '))<' ;
          END MACRO;

      MACRO TRIGGR:
          '(' BAL ')';
          RESCAN 'IF(' BAL ')' ;
          END MACRO;

      MACRO TRIGGR:
          '<*' ¬<'*+'>;
          RESCAN 'LOOP<';
          END MACRO;

      MACRO TRIGGR:
          '>UNTIL(' BAL ');';
          RESCAN ';IF(' BAL ')EXIT;>';
          END MACRO;

      MACRO TRIGGR:
          '>WHILE(' BAL ');';
          RESCAN ';IF(¬(' BAL '))EXIT;>';
          END MACRO;

      "MACRO TRIGGR:
          'D=' ATOMS ';';
          IF GLOBAL(40)=0 THEN FAIL; END IF
          GLOBAL(41)=GLOBAL(41)+1;
          RESCAN NOTRIG 'I001=D; D=';
          RESCAN ATOMS ';(P(IGL+3)=1)<OUTPUT I001,D,TX(D),V,TX(V),IA,
          IAP,LL,LA,LOF,LF,M;('' ** ' CS(GLOBAL(41)) ' ** '',12I7);>';
          WARN  'MARK--' CS(GLOBAL(41));
          END MACRO;"

      "MACRO TRIGGR:
          'P(' BAL ')=' ATOMS ';';
          I=GLOBAL(41)+1; GLOBAL(41)=I;
          RESCAN 'I001=' BAL ';(I001=991|I001=992)'
                 '<OUTPUT;('' STOP AT' CS(I) '''); STOP;>';
          RESCAN NOTRIG 'P(I001)=';
          RESCAN ATOMS ';';
          WARN 'MARK--' CS(I) ;
          END MACRO;"

      "MACRO TRIGGR:
          '(' BAL ')' ¬<'<'> ATOMS ';';
          RESCAN 'IF(' BAL ')<' ATOMS ';>';
          END MACRO;"
      MACRO TRIGGR:
          'ASSERT' ATOMS '(' BAL ');';
          RESCAN 'IF(¬('.BAL.'))'
                 '<ERR(''ASSERTION '.ATOMS.' FALSE'');STOP 12;>';
          END MACRO;

      MACRO TRIGGR:
          '+' ATOMS ';';
          RESCAN ATOMS '=' ATOMS '+1;';
          END MACRO;

      MACRO TRIGGR:
          '-' ATOMS ';';
          RESCAN ATOMS '=' ATOMS '-1;';
          END MACRO;

      MACRO TRIGGR:
          'G:' BAL ':;';
          RESCAN 'GO TO :'.BAL.':;';
          END MACRO;

      END MACRO;

MACRO SYNTAX: DCSTAT
      'COMMON'      CMSTAT |
      'DIMENSION'   DMSTAT |
      'REAL'        RLSTAT |
      'DOUBLE PRECISION' RLSTAT |
      'COMPLEX'     RLSTAT |
      'INTEGER'     IGSTAT |
      'LOGICAL'     LGSTAT |
      'EQUIVALENCE' EQSTAT |
      'DATA'        DTSTAT |
      'EXTERNAL'    EXSTAT |
      ';'                  ;
      END MACRO;

MACRO SYNTAX: STATEM
      'IF'     IFSTAT |
      'IF'     OLDIF  |
      'DO'     DOSTAT |
      'LOOP'   LPSTAT |
      'UNTIL'  ULSTAT |
      NEXIT           |
      'GO'*'TO' GOSTAT |
      'CALL'   CLSTAT |
      ASSIGN          |
      'SELECT'  SELECT |
      'OUTPUT' OTSTAT |
      'INPUT'  INSTAT |
      'REWIND' RUNIT  |
      'BACKSPACE' BUNIT |
      'END'*'FILE' EUNIT |
      'ERR'    ERSTAT |
      'ENTRY'  ENSTAT |
      'RETURN' RTSTAT |
      'STOP'*   STOPST |
      DCSTAT          |
      ';'             |
      ERRORS          ;
      GLOBAL(32)=0;
      END MACRO;

MACRO SYNTAX: IMPLIC
      'IMPLICIT ' ATOMS ';';
      OUTPUT MATCH;
      END MACRO;
```

```
MACRO SYNTAX: CMSTAT
      <1, ('/' (BLKNAM:ID.) '/') <1,DCLREF/','> / SLSH> TERMIN ;
      OUTPUT 'COMMON' MATCH ;
      END MACRO;

MACRO SYNTAX: SLSH
      '/';
      RESCAN MATCH;
      END MACRO;

MACRO SYNTAX: DMSTAT
      DCLGEN ;
      OUTPUT 'DIMENSION' MATCH ;
      END MACRO;

MACRO SYNTAX: RLSTAT
      ('*4'* | '*8'*) DCLGEN ;
      OUTPUT 'REAL' MATCH ;
      END MACRO;

MACRO SYNTAX: IGSTAT
      ('*2'* | '*4'*) DCLGEN ;
      OUTPUT 'INTEGER' MATCH ;
      END MACRO;

MACRO SYNTAX: LGSTAT
      ('*1'* | '*4'*) DCLGEN ;
      OUTPUT 'LOGICAL' MATCH ;
      END MACRO;

MACRO SYNTAX: EQSTAT
      <1, '(' DCLREF ',' DCLREF ')' / ',' > ';';
      OUTPUT 'EQUIVALENCE' MATCH ;
      END MACRO;

MACRO SYNTAX: DTSTAT
      <1, <1, DCLREF / ','> '/' <1, DTITEM/ ','> '/' / ','> ';';
      OUTPUT 'DATA' MATCH ;
      END MACRO;

MACRO SYNTAX: DTITEM
      (NUM'*')< ('+'|'-')<FLT|NUM>|STCONS|'T'|'.TRUE.'|'F'|'.FALSE.'> ;
      ANSWER MATCH ;
      END MACRO;

MACRO SYNTAX: EXSTAT
      <1, ID/',' > TERMIN;
      OUTPUT 'EXTERNAL' MATCH;
      END MACRO;

MACRO SYNTAX: DCLGEN
      <1, MAIN:<ID('('<1,<NUM|D2:ID.>/',','>')')> DATA:('/' BAL '/') /
           COMMA:','> TERMIN ;
      L=ID(*);
      FOR I=1 TO L DO
          ANSWER MAIN(I) COMMA(I);
          IF DATA(I) THEN
              RESCAN 'DATA' ID(I) DATA(I) ';';
          END IF
      END FOR
      ANSWER ';';
      END MACRO;

MACRO SYNTAX: DCLREF
      ID ( '(' <1,NUM/','> ')' ) ;
      ANSWER MATCH;
      END MACRO;
```

```
MACRO SYNTAX: FLT
      (NUM ¬<' '>)  '.'
          (¬<' '> NUM2:NUM) (<'E'|'D'> ('+'|'-') NUM3:NUM);
      IF ¬(NUM | NUM2) THEN FAIL; END IF
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: LABGEN
      ':' BAL ':' ;
      SYMBOL LABEL(*,2) ;
      K=LABEL@BAL; L1=LABEL(K,1);
      IF L1=0 THEN
          L1=GLOBAL(31)+10; GLOBAL(31)=L1; LABEL(K,1)=L1;
      END IF
      OUTPUT CS(L1) 'CONTINUE;';
      GLOBAL(32)=K;
      END MACRO;

MACRO SYNTAX: IFSTAT
      '(' LOGNEG ')<' ;
      L1=GLOBAL(31)+10; GLOBAL(31)=L1; L2=L1; FLAG=0; FLAG2=0;
      OUTPUT 'IF(' LOGNEG ')GOTO' CS(L1+1) ';' ;
      SCAN SGROUP RBRACK;
      LOOP
          SCAN <'*('|'ELSEIF('> LOG2:LOGNEG ')<' ;
          IF ¬LOG2 THEN EXIT; END IF  FLAG=1; FLAG2=1;
          OUTPUT 'GOTO' CS(L1+2) ';' CS(L2+1) 'CONTINUE;' ;
          L2=GLOBAL(31)+10; GLOBAL(31)=L2;
          OUTPUT 'IF(' LOG2 ')GOTO' CS(L2+1) ';' ;
          SCAN GROUP2: SGROUP RBRACK;
      END LOOP
      SCAN ELSTAT: 'ELSE<' ;
      IF ELSTAT THEN
          FLAG=1; FLAG2=0;
          OUTPUT 'GOTO' CS(L1+2) ';' CS(L2+1) 'CONTINUE;' ;
          SCAN GROUP3: SGROUP RBRACK;
      END IF
      IF FLAG THEN
          IF FLAG2 THEN OUTPUT CS(L2+1) 'CONTINUE;'; END IF
          OUTPUT CS(L1+2);
      ELSE
          OUTPUT CS(L1+1);
      END IF
      OUTPUT 'CONTINUE;' ;
      END MACRO;

MACRO SYNTAX: OLDIF
      '(' LOGEXP ')';
      OUTPUT 'IF(' LOGEXP ')';
      END MACRO;

MACRO SYNTAX: ASSIGN
      REF '=' ARIEXP TERMIN | LOGREF '=' LOGEXP TERMIN ;
      OUTPUT MATCH;
      END MACRO;

MACRO SYNTAX: SELECT
      'USING' ID '<';
      SYMBOL CASYM(*); MARK CASYM;
      ENDLAB=GLOBAL(31)+10; GLOBAL(31)=ENDLAB;
      OUTPUT 'GO TO' CS(ENDLAB) ';';
      MAXC=0;
      LOOP
         SCAN 'CASE' <1, NUM/ ','> BRAC:'<';
         IF ¬BRAC THEN EXIT; END IF
         LOC=GLOBAL(31)+10; GLOBAL(31)=LOC;
         FOR I=1 TO NUM(*) DO
            CP=CASYM.LOCAL@NUM(I);
```

```
             IF CASYM(CP)¬=0 THEN WARN '** CASE LABEL'NUM(I)'USED TWICE';
                    END IF
             CASYM(CP)=LOC;
             K=CN(NUM(I)); IF (K>MAXC) THEN MAXC=K; END IF
          END FOR
          OUTPUT CS(LOC) 'CONTINUE;';
          SCAN SGROUP RBRACK;
          OUTPUT 'GO TO' CS(ENDLAB+2) ';';
       END LOOP
”NOW OUTPUT THE COMPUTED GO TO STATEMENT”

       OUTPUT CS(ENDLAB)
              ” 'IF('ID'.GT.'CS(MAXC)')GO TO'CS(ENDLAB+1)';' ”  ”CDC”
              'GO TO (';
       STRING COMMA;
       FOR I=1 TO MAXC DO
          CP=CASYM.LOCAL.NONEW@CS(I);
          IF CP¬=0 THEN
             OUTPUT COMMA CS(CASYM(CP));
          ELSE
             OUTPUT COMMA CS(ENDLAB+1);
             WARN ' ** CASE' CS(I) 'NOT REFERENCED **';
          END IF
          COMMA=',';
       END FOR
       OUTPUT '),' ID ';' CS(ENDLAB+1) 'CONTINUE;';
       SCAN CELSE: 'CASE ELSE<';
       IF CELSE THEN
          SCAN SGROUP RBRACK;
       ”ELSE
          PUT ERROR MESSAGE AND STOP STATEMENT HERE”
       END IF
       SCAN RBRACK;
       OUTPUT CS(ENDLAB+2) 'CONTINUE;';
       DROP CASYM;
       END MACRO;
MACRO SYNTAX: DOSTAT
       PARMS:<ID '=' START:<ID2:ID.|NUM> ',' FINISH:<ID3:ID.|NUM2:NUM>
                               ( ','   STEP:<ID4:ID.|NUM3:NUM> )> '<' ;
       SYMBOL LABEL(*,2) ;
       K=GLOBAL(32);
       IF K=0 THEN
           L1=GLOBAL(31)+10; GLOBAL(31)=L1;
       ELSE
           LABEL(K,2)=1;
           L1=LABEL(K,1);
           GLOBAL(32)=0;
       END IF
       LSAVE=GLOBAL(33); GLOBAL(33)=L1;
       OUTPUT 'DO' CS(L1+1) PARMS ';';
       SCAN SGROUP RBRACK;
       OUTPUT CS(L1+1) 'CONTINUE;' CS(L1+2) 'CONTINUE;';
       GLOBAL(33)=LSAVE;
       END MACRO;
MACRO SYNTAX: LPSTAT
       '<' ;
       SYMBOL LABEL(*,2) ;
       K=GLOBAL(32);
       IF K=0 THEN
           L1=GLOBAL(31)+10; GLOBAL(31)=L1;
       ELSE
           LABEL(K,2)=1;
           L1=LABEL(K,1);
           GLOBAL(32)=0;
       END IF
```

```
      LSAVE=GLOBAL(33); GLOBAL(33)=L1;
      OUTPUT CS(L1+1) 'CONTINUE;';
      SCAN SGROUP RBRACK;
      OUTPUT 'GO TO' CS(L1+1) ';' CS(L1+2) 'CONTINUE;';
      GLOBAL(33)=LSAVE;
      END MACRO;

MACRO SYNTAX: ULSTAT
      '('LOGEXP ')<' ;
      SYMBOL LABEL(*,2) ;
      K=GLOBAL(32);
      IF K=0 THEN
          L1=GLOBAL(31)+10; GLOBAL(31)=L1;
      ELSE
          LABEL(K,2)=1;
          L1=LABEL(K,1);
          GLOBAL(32)=0;
      END IF
      LSAVE=GLOBAL(33); GLOBAL(33)=L1;
      OUTPUT CS(L1+1) 'IF(' LOGEXP ')GO TO' CS(L1+2) ';' ;
      SCAN SGROUP RBRACK;
      OUTPUT 'GO TO' CS(L1+1) ';' CS(L1+2) 'CONTINUE;';
      GLOBAL(33)=LSAVE;
      END MACRO;

MACRO SYNTAX: NEXIT
      <A:'NEXT' | B:'EXIT'> (':' BAL ':') ';' ;
      SYMBOL LABEL(*,2) ;
      IF A THEN J=1; ELSE J=2; END IF
      IF BAL THEN
          K=LABEL.NONEW@BAL;
          IF K=0 THEN GO TO ERROR; END IF
          IF LABEL(K,2)¬=1 THEN GO TO ERROR; END IF
          OUTPUT 'GO TO' CS(LABEL(K,1)+J) ';';
      ELSE
          I=GLOBAL(33);
          IF I¬=0 THEN
              OUTPUT 'GO TO' CS(I+J) ';' ;
          ELSE
              ERROR: WARN '** STATEMENT' SOURCE 'DELETED**';
          END IF
      END IF
      END MACRO;

MACRO SYNTAX: LABREF
      ':' BAL ':' ;
      SYMBOL LABEL(*,2) ;
      K=LABEL@BAL; L1=LABEL(K,1);
      IF L1=0 THEN
          L1=GLOBAL(31)+10; GLOBAL(31)=L1; LABEL(K,1)=L1;
      END IF
      ANSWER CS(L1);
      END MACRO;

MACRO SYNTAX: GOSTAT
      LABREF ';' ;
      OUTPUT 'GO TO' LABREF ';';
      END MACRO;

MACRO SYNTAX: CLSTAT
      ID ('('<1,<ARIEXP|LOGEXP|STCONS> /','> <0, ',&' LABREF> ')') ';' ;
      OUTPUT 'CALL' MATCH ;
      END MACRO;
```

```
MACRO SYNTAX: OTSTAT
      ( '(' UNIT:<ID|NUM> ')' ) IOLIST ';' ( FMTEXP TERMIN ) ;
      IF ¬UNIT THEN UNIT='ILU'; END IF
      OUTPUT 'WRITE(' UNIT ;
      IF FMTEXP THEN
          L1=GLOBAL(31)+10; GLOBAL(31)=L1;
          OUTPUT ',' CS(L1) ;
      END IF
      OUTPUT ')' IOLIST ';';
      IF FMTEXP THEN
          OUTPUT CS(L1) 'FORMAT' FMTEXP ';' ;
      END IF
      END MACRO;

MACRO SYNTAX: INSTAT
      ( '(' UNIT:<ID|NUM> ')' ) IOLIST ';' ( FMTEXP TERMIN ) ;
      IF ¬UNIT THEN UNIT='IIU'; END IF
      OUTPUT 'READ(' UNIT ;
      IF FMTEXP THEN
          L1=GLOBAL(31)+10; GLOBAL(31)=L1;
          OUTPUT ',' CS(L1) ;
      END IF
      OUTPUT ')' IOLIST ';';
      IF FMTEXP THEN
          OUTPUT CS(L1) 'FORMAT' FMTEXP ';' ;
      END IF
      END MACRO;

MACRO SYNTAX: IOLIST
      <0,< REF ¬<'='> | '(' IOLIST ',' ID '='
                 <NUM|ID> ',' <NUM|ID> (',' <NUM|ID>) ')'>/ ','> ;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: FMTEXP
      '(' (<1,'/'>(',')) <0, FMTITM/FCOMSL> ((',')<1,'/'>) ')' ;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: FCOMSL
      SL1:<0, '/'> COMMA:(',') SL2:<0, '/'> ;
      IF ¬SL1 & ¬SL2 THEN
          IF ¬COMMA & GLOBAL(35)=0 THEN FAIL; END IF
          ANSWER ',' ;
      ELSE
          ANSWER SL1.SL2 ;
      END IF
      END MACRO;

MACRO SYNTAX: FMTITM
      (NUM) FMTEXP |
      STCONS | N1:NUM 'X' | 'T'*N2:NUM |
      (N3:NUM)<'A'*|'R'*|'I'*|'L'*|'Z'*> N4:NUM |
      (N5:NUM 'P'*)(N6:NUM) <'D'*|'E'*|'F'*|'G'*> N7:NUM '.' N8:NUM ;
      IF STCONS THEN GLOBAL(35)=1; ELSE GLOBAL(35)=0; END IF
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: RUNIT
      <ID | NUM> TERMIN ;
      OUTPUT 'REWIND' MATCH ;
      END MACRO;

MACRO SYNTAX: BUNIT
      <ID | NUM> TERMIN ;
      OUTPUT 'BACKSPACE' MATCH ;
      END MACRO;
```

```
MACRO SYNTAX: EUNIT
      <ID | NUM> TERMIN ;
      OUTPUT 'END FILE' MATCH ;
      END MACRO;

MACRO SYNTAX: ERSTAT
      '(' (A:<1,'/'>(',')) <1,<STCONS | IOLIST ':' FMTITM> / FCOMSL>
              ((',') B:<1,'/'>) ');' ;
      L1=GLOBAL(31)+10; GLOBAL(31)=L1;
      OUTPUT 'WRITE(ILU,' CS(L1) ')';
      L=IOLIST(*); RFLAG=0;
      FOR I=1 TO L DO
          IF IOLIST(I) THEN
              IF RFLAG THEN OUTPUT ','; END IF
              OUTPUT IOLIST(I) ;
              RFLAG=1;
          END IF
      END FOR
      OUTPUT ';' CS(L1) 'FORMAT(' A '11H ERROR****  ,'
              STCONS(1) FMTITM(1);
      FOR I=2 TO L DO
          OUTPUT FCOMSL(I-1) STCONS(I) FMTITM(I) ;
      END FOR
      OUTPUT B ');';
      END MACRO;

MACRO SYNTAX: ENSTAT
      ID ('(' ARGLST ')') TERMIN ;
      OUTPUT 'ENTRY' MATCH;
      END MACRO;

MACRO SYNTAX: RTSTAT
      (NUM) TERMIN;
      OUTPUT 'RETURN' NUM ';' ;
      END MACRO;

MACRO SYNTAX: STOPST
      (NUM) TERMIN;
      OUTPUT 'STOP' NUM ';' ;
      END MACRO;

MACRO SYNTAX: DCWARN
      CHAR; RESCAN CHAR;
      WARN '** DECLARE STATEMENT OUT OF ORDER **';
      END MACRO;

MACRO SYNTAX: ERRORS
     ¬<'>' | 'END;'>  ATOMS ';';
     WARN '** STATEMENT -- ' SOURCE ' -- PASSED';
     OUTPUT MATCH;
     END MACRO;

MACRO SYNTAX: STCONS
      STR;
      STRING S; S=STR;
      L=LENGTH(S)-2;
      S=SUBS(S,2,L);
      ANSWER CS(L).'H';

      LOOP
          I=INDEX(S,'''''');
          IF I=0 THEN EXIT; END IF;
          IF SUBS(S,I+1)='''''' THEN
              ANSWER .SUBS(S,1,I);
              S=SUBS(S,I+2);
          ELSE
              ANSWER .SUBS(S,1,I-1).'''''';
              S=SUBS(S,I+1);
          END IF
      END LOOP
      ANSWER .S;
      END MACRO;

MACRO SYNTAX: LOGEXP
      <1, OREXP / OR >;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: OREXP
      <1, ANDEXP / AND >;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: ANDEXP
      RELATN | (NOT) LOGIC | '(' LOGEXP ')' | NOT '(' LOGNEG ')' ;
      IF ¬LOGNEG THEN
          ANSWER MATCH;
      ELSE
          ANSWER '(' LOGNEG ')' ;
      END IF
      END MACRO;

MACRO SYNTAX: LOGIC
      LOGREF | '.TRUE.' | '.FALSE.' ;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: LOGREF
      REF;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: RELATN
      (NOT) A:ARIEXP RELOPF B:ARIEXP;
      IF NOT THEN
          RELOPF=SUBS('.EQ.NE.GT.GE.LT.LE.',
                INDEX('.NE.EQ.LE.LT.GE.GT.',RELOPF),4);
      END IF
      ANSWER A RELOPF B;
      END MACRO;

MACRO SYNTAX: RELOPF  ″NOTE-TWO TEMPORARY CHANGES IN THIS MACRO ***″
      '=' | '¬' | '>=' | '>' | '<=' | '<' ;
      ANSWER '.'.SUBS('EQNEGTGELELE',INDEX('= ¬=> >=< <=',MATCH),2).'.';
      END MACRO;

MACRO SYNTAX: NOT
      '¬' ¬<'='>;
      ANSWER '.NOT.';
      END MACRO;

MACRO SYNTAX: AND
      '&';
      ANSWER '.AND.';
      END MACRO;

MACRO SYNTAX: OR
      '|' ¬<'|'>;
      ANSWER '.OR.';
      END MACRO;

MACRO SYNTAX: REF
      ID ( '(' <1,ARIEXP/ ',' > ')' ) ;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: LOGNEG
      <1, ORNEG/NOR>;
      ANSWER MATCH;
      END MACRO;
```

```
MACRO SYNTAX: NOR
      OR;
      ANSWER '.AND.';
      END MACRO;

MACRO SYNTAX: ORNEG
      <1, NOTINS ANDEXP / NAND>;
      IF (ANDEXP(*)>1) THEN
          ANSWER '(' MATCH ')' ;
      ELSE
          ANSWER MATCH;
      END IF
      END MACRO;

MACRO SYNTAX: NAND
      AND;
      ANSWER '.OR.' ;
      END MACRO;

MACRO SYNTAX: NOTINS
      (NOT) ;
      IF ¬NOT THEN RESCAN '¬'; END IF
      END MACRO;

MACRO SYNTAX: ARIEXP
      ('+'|'-') <1, TERM / '+'|'-'>;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: TERM
      <1, FACTOR / '*'|'/'>;
      ANSWER MATCH;
      END MACRO;

MACRO SYNTAX: FACTOR
      < REF |  FLT | NUM | '(' ARIEXP ')' > ('**' FACTOR);
      ANSWER MATCH;
      END MACRO;

MACRO TRIGGR:
      '$(' EVALU ')';
      RESCAN EVALU;
      END MACRO;

MACRO SYNTAX: EVALU
      ('+'| PREFIX:'-')<1, <1, EFACTR/TIMES:'*'|'/'>/
                          PLUS:'+'|'-'> ;
      LIM=EFACTR(*);
      FOR II=1 TO LIM DO
          ELIM=EFACTR(*,II);
          F=CN(EFACTR(1,II));
          FOR I=2 TO ELIM DO
              FACTOR=CN(EFACTR(I,II));
              IF TIMES(I-1,II) THEN
                  F=F*FACTOR;
              ELSE
                  F=F/FACTOR;
              END IF
          END FOR
          IF II=1 THEN
              IF PREFIX THEN F=-F; END IF
              SUM=F;
          ELSEIF PLUS(II-1) THEN
              SUM=SUM+F;
          ELSE
              SUM=SUM-F;
          END IF
      END FOR
      ANSWER CS(SUM);
      END MACRO;
```

```
MACRO SYNTAX: EFACTR
      ('-') NUM | '(' EVALU ')';
      IF NUM THEN ANSWER MATCH;
      ELSE ANSWER EVALU; END IF
      END MACRO;
```